D

# NAVAL POSTGRADUATE SCHOOL
# Monterey, California

**THESIS**

**NODE TO PROCESSOR ALLOCATION FOR LARGE GRAIN DATA FLOW GRAPHS IN THROUGHPUT-CRITICAL APPLICATIONS**

by

John P. Cardany

June 1994

Thesis Advisor: Shridhar B. Shukla

Approved for public release; distribution is unlimited.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION/AVAILABILITY OF REPORT |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S) | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School; 6b. OFFICE SYMBOL (If applicable) ECE | 7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School |
| 6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000 | 7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c. ADDRESS (City, State, and ZIP Code) | 10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

11. TITLE (Include Security Classification)
Node to Processor Allocation for Large Grain Data Flow Graphs in Throughput-Critical Applications (U)

12. PERSONAL AUTHOR(S)
Cardany, John Paul

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year,Month,Day) | 15. PAGE COUNT |
|---|---|---|---|
| Masters Thesis | FROM ____ TO ____ | June 1994 | 95 |

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Revolving Cylinder (RC), Start after Finish (SAF), Large Grain Data Flow (LGDF) Graphs , Node Allocation |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis describes the issues involved in node allocation for a Large Grain Data Flow (LGDF) model used in Navy signal processing applications. In the model studied, nodes are assigned to processors based on load balancing, communication / computation overlap, and memory module contention. Current models using the Revolving Cylinder (RC) technique for LGDF graph analysis do not adequately address node allocation. Thus, a node to processor allocation component is added to a computer simulator of an LGDF graph model. It is demonstrated that the RC technique, when proper node allocation is taken into account, can improve overall throughput as compared to the First-Come-First-Served (FCFS) technique for high communication/computation costs.

| | | |
|---|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ X] UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION Unclassified | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL Shukla, Shridhar B. | 22b. TELEPHONE (Include Area Code) (408) 656-2764 | 22c. OFFICE SYMBOL EC/Sh |

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603 Unclassified

# Node to Processor Allocation for Large Grain Data Flow Graphs in Throughput-Critical Applications

by

**John Paul Cardany**
**Lieutenant, United States Navy**
**B.S., University of Washington, 1987**

Submitted in partial fulfillment of the requirements for the degree of

**MASTER OF SCIENCE IN ELECTRCAL ENGINEERING**

from the

**NAVAL POSTGRADUATE SCHOOL**

**June 1994**

Author: [illegible]
John P. Cardany

Approved by: [illegible]
Shridhar B. Shukla, Thesis Advisor

Amr Zaky
Amr Zaky, Second Reader

[illegible]
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

## ABSTRACT

This thesis describes issues involved in node allocation for a Large Grain Data Flow (LGDF) model used in Navy signal processing applications. In the model studied, nodes are assigned to processors based on load balancing, communication / computation overlap, and memory module contention. Current models [illegible]g the Revolving Cylinder (RC) technique for LGDF graph analysis do not adequately address node allocation. Thus, a node to processor allocation component is added to a computer simulator of an LGDF graph model. It is demonstrated that the RC technique, when proper node allocation is taken into account, can improve overall throughput as compared to the first-come-first-served (FCFS) technique for high communication / computation costs.

## TABLE OF CONTENTS

# I. INTRODUCTION

The Revolving Cylinder (RC) technique [Ref. 1] was developed as an attempt to enhance throughput over the First-Come-First-Served (FCFS) technique for dispatching nodes for communication intensive applications. A computer programmed simulator based on the Department of the Navy's AN/UYS-2 Digital Signal Processing System, also known as the Enhanced Modular Signal Processor (EMSP) [Ref. 2], was developed to evaluate the RC techniques with respect to such machines. In this thesis, a node to processor allocation component has been added to the simulator.

## A. BACKGROUND

Large Grain Data Flow (LGDF) graphs are particularly suited to describing applications where large amounts of data are generated and require predictable, periodic processing. Thus, LGDF graphs are often used to model signal processing applications with specific throughput requirements. LGDF graph execution can be carried out using a balance of compile-time and run-time decisions in order to achieve the most efficient throughput. Digital signal processing (DSP) applications lend themselves easily to compile-time analysis because DSP applications are very specific in the computation required for each node [Ref. 3]. The AN/UYS-2 programs use large grain data flow execution as their run-time environment and thus can be modeled using an LGDF graph representation.

For an LGDF graph receiving periodic input data, FCFS cannot provide uniform throughput under high loads because the nodes receiving external data become ready independent of other nodes in the graph and thus the nodes higher in the graph become ready before the lower nodes in the graph. This results in system congestion and causes a decrease in throughput. The RC technique adds graph dependencies to the nodes in the graph thus reducing or eliminating this congestion to ensure a more uniform throughput.

The FCFS scheduling technique places nodes into the system based on when the nodes are ready. Thus FCFS cannot benefit from compile-time efforts in scheduling nodes nor does it bind nodes to specific processors for execution. In previous applications of the RC technique, graph dependencies were added at compile-time based on node allocation that was performed randomly. Performance with this random allocation, however, was poorer than that provided by FCFS. Thus, in order to ensure that RC facilitates better performance than FCFS, it is necessary to modify the generation of graph dependencies using the RC technique based on the node to processor allocation.

## B. THESIS SCOPE AND CONTRIBUTION

This thesis describes an algorithm for allocation of nodes to processors for an LGDF graph. A real application modeled as an LGDF graph is studied, based on a signal correlator graph representing an actual application running on the AN/UYS-2. Results are generated using the node allocation program as well as previously developed software and comparisons made between the First-Come-First-Served (FCFS) technique and the Revolving Cylinder (RC) technique.

## C. THESIS ORGANIZATION

Chapter II describes the issues involved in node allocation for improving the performance of the LGDF. Included are the problems existing with current allocation methods and the issues addressed as a result of these deficiencies. Chapter III gives a description of the algorithms used in the node allocation program as they relate to the issues in Chapter II. Chapter IV presents the analysis of data generated from several scheduling methods. Chapter V summarizes the results, presents conclusions drawn from the data analysis, and provides topics of further study.

# II. ISSUES IN ALLOCATION OF NODES

There are several issues relating to the task of node allocation. In the model discussed in this thesis, nodes are assigned to processors based on several factors, such as load balancing, overlap of communication and computation, and contention between nodes for memory modules. Each of these is important and a delicate balance between these factors must be accomplished in order to achieve maximum utilization of the processors.

## A. PROBLEMS WITH CURRENT ALLOCATION

Node allocation in the general sense refers to the binding of nodes to specific processors for execution based on certain criteria. Allocation is separate from scheduling which refers to determining the time at which the node executes on the processor to which it is allocated. Without proper node allocation, the processors cannot execute at their most efficient level, and throughput for the data flow graph is reduced as a result. To demonstrate this, the programs in Appendix A and [Ref. 4] were used on a test data flow graph, illustrated in Figure 2.1, to allocate nodes and simulate the data flow graph.

The graph shows two input/output processors and 13 nodes. Even numbered nodes were assumed to have two times the number of execution cycles as odd numbered nodes. Each individual queue's produce, consume, write, and read amounts were considered equal; however these values differed over different queues. These are the values shown on the queues in Figure 2.1. The queue capacity was equal to eight times the queue threshold. The simulation was run with three processors and no setup or breakdown latency for the nodes was assumed. In addition, the scheduler latency was zero and the communication time for one word was five cycles. The simulation was run first without node allocation, i.e., the nodes are assigned to processors without regard for satisfying the criteria described above, and then with proper node allocation. In the first case, the nodes were allocated dynamically at run-time based on which node was ready and which processor was free. In the second case, the nodes were allocated statically at compile-time based on load balancing, queue contention, and memory module contention. The results are compared in the graph of Figure 2.2. Note the lower utilization rate of the execution unit of the processor for the simulation without node allocation as well as the lower throughput.

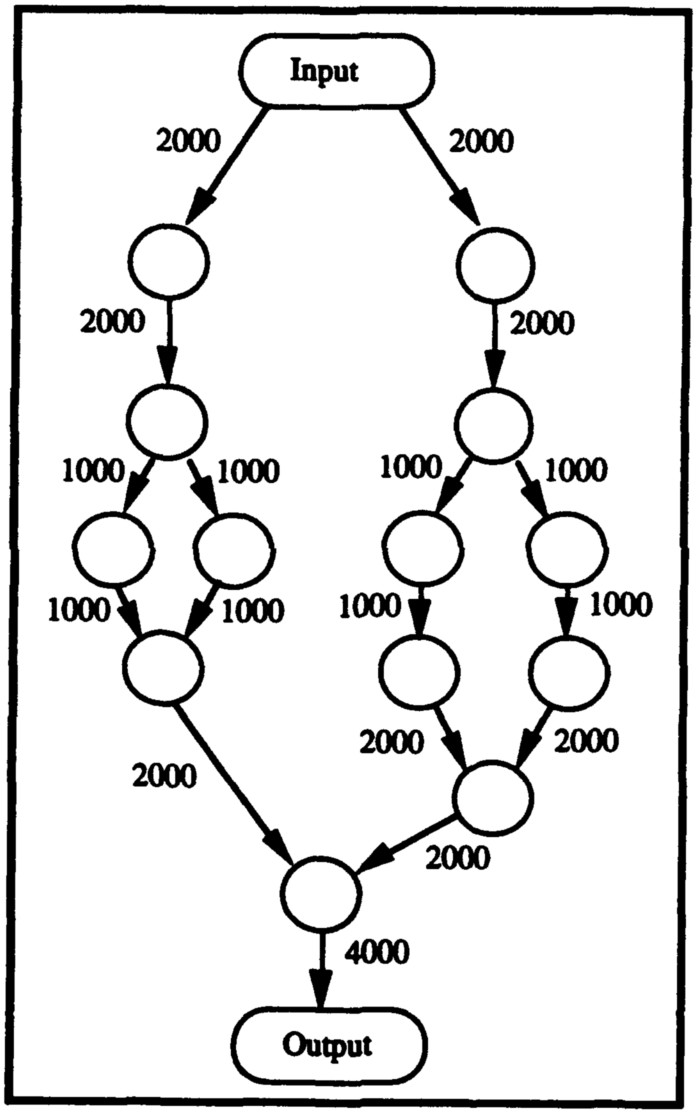

**Figure 2.1. Test Data Flow Graph**

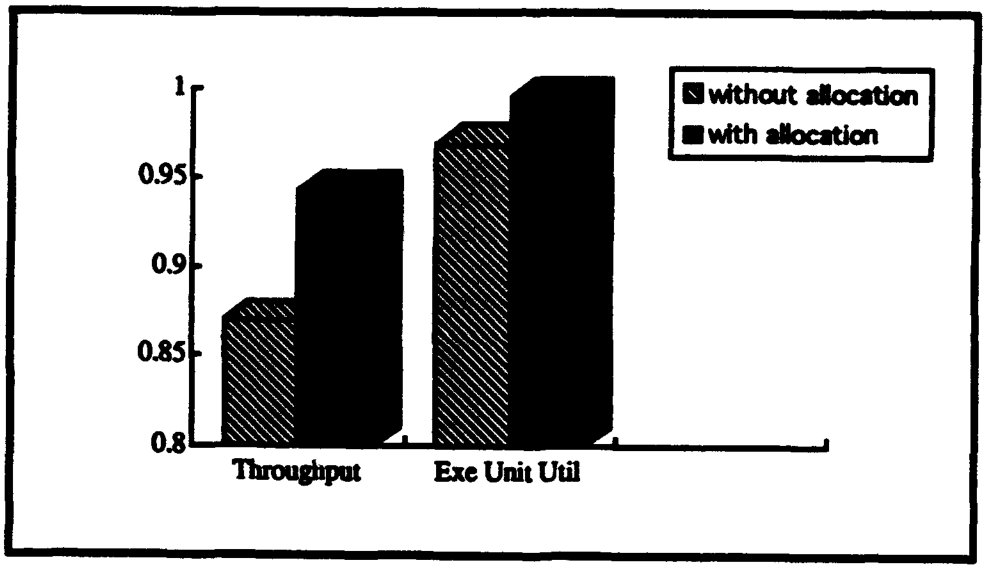

**Figure 2.2. Improvement With Allocation Over No Allocation**

## B. ISSUES ADDRESSED

### 1. Load Balancing

In order to ensure the processors are being fully utilized, it is important to ensure that the nodes executing across processors are balanced with respect to execution and/or communication times. Since the emphasis of the node allocation algorithms is based upon maximization of the execution unit utilization, load balancing for the processors will focus mainly on the execution time of the nodes. Load balancing is achieved by statically assigning nodes to processors based on execution times of the nodes, attempting to maintain the same number of execution cycles per processor.

### 2. Overlap

Overlap of communication and computation is important to the LGDF model of computation. The system contains both a control unit and an execution unit per processor. It is desirable to utilize both of these units in a way that permits use of the execution unit to the fullest extent possible. This is achieved by overlap of communication and computation. There are two conditions which must be met for nodes to overlap sufficiently such that the execution unit is utilized to the fullest extent possible. For two nodes j and j+1, where node j executes on the processor before node j+1, the following two conditions should exist:

$$\text{execution}_j \geq \text{setup}_{j+1}. \qquad (1)$$

and

$$\text{breakdown}_j \leq \text{execution}_{j+1} \qquad (2)$$

Ideally, perfect overlap of communication and computation is desired, such as that shown in Figure 2.3.

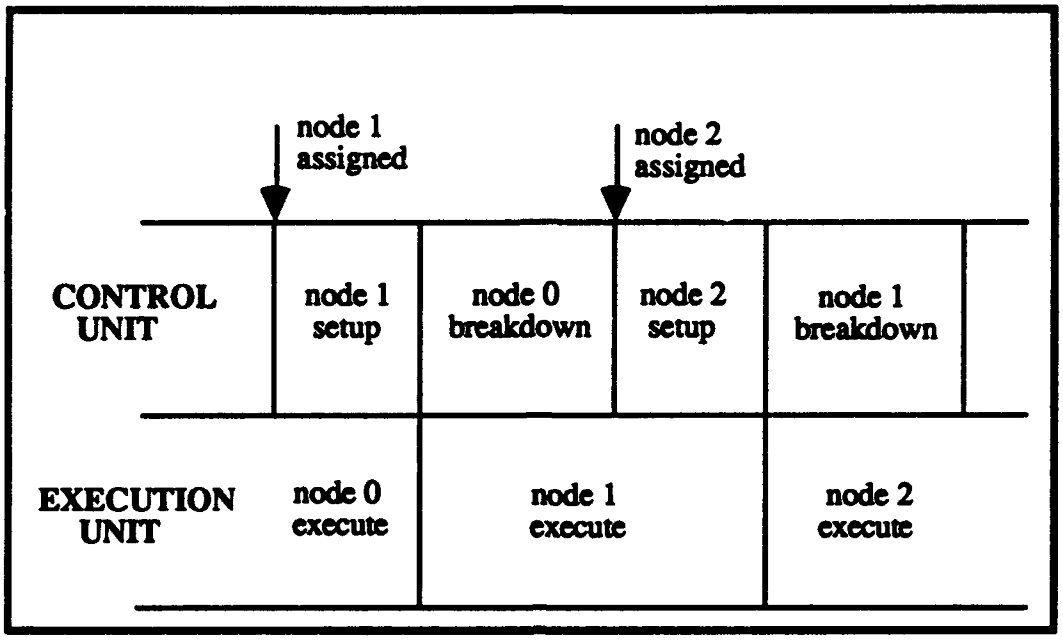

**Figure 2.3. Ideal Communication / Computation Overlap [Ref. 2]**

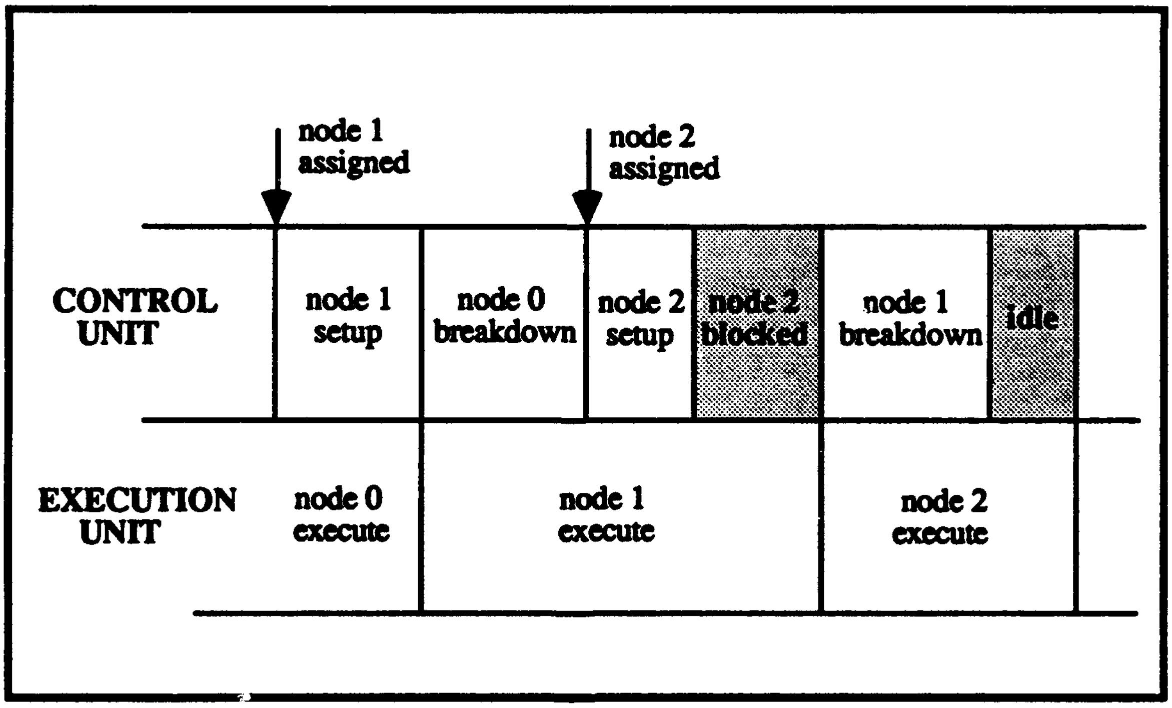

**Figure 2.4. Typical Communication / Calculation Overlap [Ref. 2]**

In this figure, it is assumed that node 0 has been executing for some time before node 1 is assigned. Here there are no idle or blocked cycles on the processor, since nodes can progress immediately from input (setup) to execution to output (breakdown). Note that both condition (1) and condition (2) are met for all nodes and the execution unit is operating continuously. This, however, is not always the case in reality, as shown in Figure 2.4.

In Figure 2.4, there is contention for the execution unit since node 2 has completed input but cannot progress to the execution unit because node 1 is still executing. This results in blocked cycles until node 1 has finished executing. In addition, idle cycles

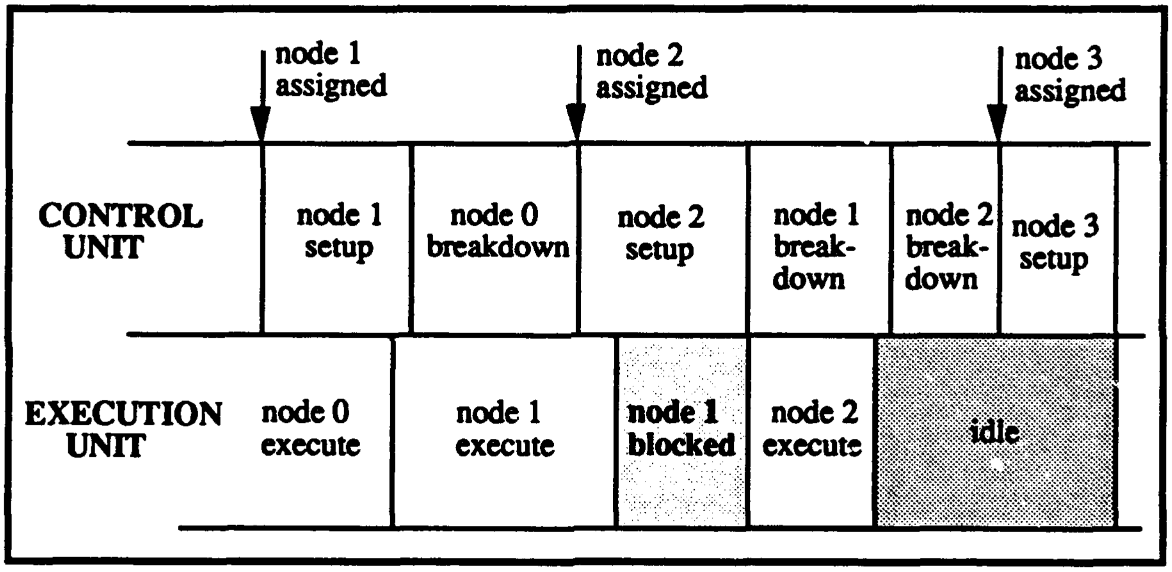

**Figure 2.5 Poor Communication / Computation Overlap [Ref. 2]**

may also exist, such as above, where node 1 has finished breakdown but node 2 is still executing and does not yet require the control unit. It is desirable to limit the blocked and idle times to maximize overlap wherever possible. In addition, a situation may also occur where there is poor overlap, such as that in Figure 2.5.

In this figure node 2 has not completed setup after node 1 finishes execution, and node 1 must therefore wait for access to the control unit, creating idle cycles on the execution unit. In addition, node 1's breakdown is longer than node 2's execution. This results in additional idle cycles, since node 2 must breakdown and node 3 must setup before the execution unit is utilized again.

### 3. Memory Contention

The memory modules are the representation for the system memory [REF 1]. Each processor must address memory modules to transfer data to or from a node during a read or write operation, respectively. Each queue in the data flow graph is assigned a memory module either by the user or arbitrarily by the scheduler. Only one processor can access a given memory module at any one time. It is possible however, for a processor to be accessing a memory module (either reading or writing) while another processor is attempting to utilize the same memory module. This is memory contention. Thus the processor which is attempting to access the memory module must wait until the memory module is free. This delays the completion of the graph and affects throughput. Memory contention can be reduced or avoided by ensuring that sufficient memory modules exist to fulfill the requirements of all the queues of the graph. Alternatively, queues can be mapped on the available memory modules such that this contention is minimized.

## C. WRAP-AROUND

Wrap-around is a technique used to maximize the overlap as permitted by the RC approach by statically 'wrapping' the breakdown time of the last node to the idle or blocked time at the head of a cylinder. An example will better illustrate this principle. Figure 2.5 is a static representation of a cylinder with three nodes on a single processor.

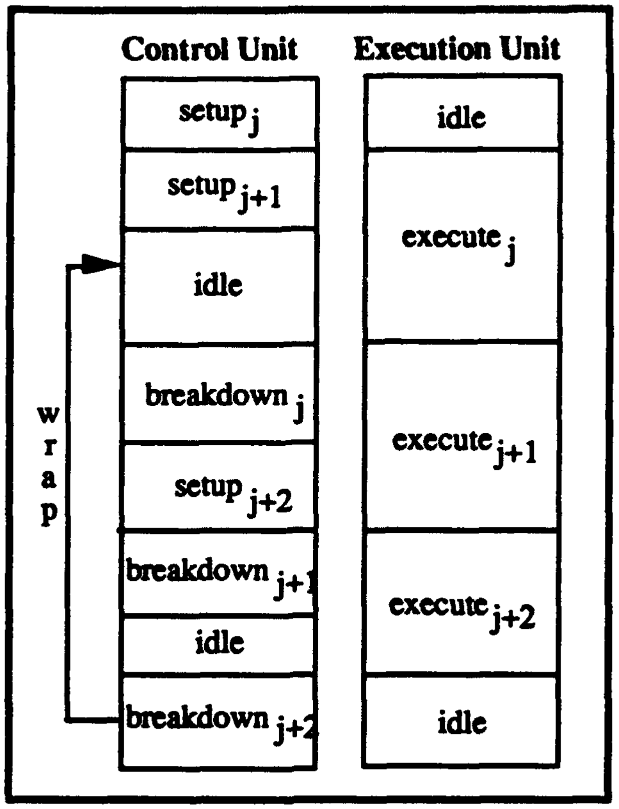

**Figure 2.5. Cylinder Without Wrap-Around**

Both control unit and execution unit are shown. Note that because node j+1's setup time is shorter than node j's execution time, blocked cycles result on the control unit. If node j+2's breakdown time is sufficiently short such that node j+2's breakdown time can be placed in the blocked cycle time, the cylinder length is reduced and the number of blocked cycles are reduced, increasing control unit utilization. The resultant cylinder is shown in Figure 2.6.

Note that the iteration index of node j+2's breakdown has changed to indicate that the breakdown is now from a previous iteration. The goal of wrap-around is to attempt to shorten the length of the cylinder by an amount equal to the length of the breakdown time of the last node without extending the length of the execution unit.

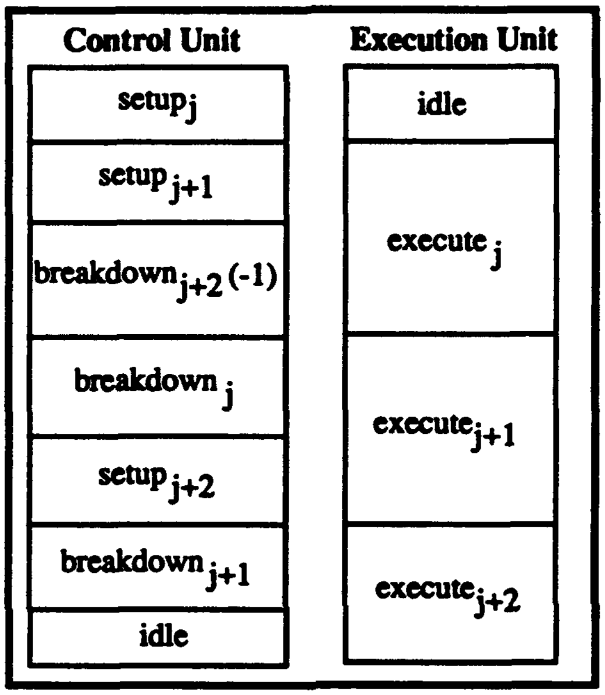

**Figure 2.6. Cylinder With Wrap-Around**

In general, for one or two nodes j (and j+1) executing on a processor where node j executes before node j+1, wrap-around is possible if:

$$\text{setup}_{j+1} + \text{breakdown}_j + \text{breakdown}_{j+1} \leq \text{execution}_j + \text{execution}_{j+1}$$

as long as at least condition (1) is satisfied.

For three or more nodes on a processor, the general case becomes more complicated, because there is a potential for the third node's setup time to occur during the second node's execution time. In this case, wrap-around is dependent on which condition(s) listed above is (are) satisfied.

Let there exist nodes j, j+1, j+2, and j+N, where j is the first node, j+1 is the second node, j+2 is the third node, and j+N is the last node on a processor with N nodes. For exactly three nodes on a processor, j+N and j+2 are synonymous. There are three cases for wrap-around:

Case 1: only condition (1) is satisfied. Figure 2.7 illustrates this case. Here, since node j's breakdown is greater than node j+1's execution time, node j+2's setup time cannot be overlapped with node j+1's execution time. Thus, wrap-around is possible if:

$$\text{setup}_{j+1} + \text{breakdown}_{j+N} \leq \text{execution}_j$$

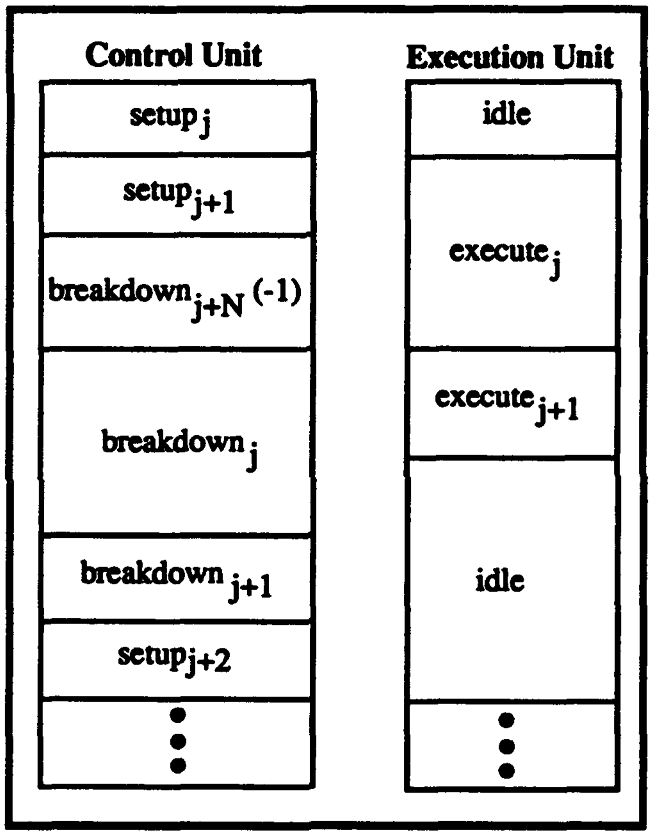

**Figure 2.7. Wrap-Around (Case 1)**

Case 2: only condition (2) is satisfied. Figure 2.8 illustrates this. For this condition wrap-around cannot occur at all, since doing so would extend the length of the execution unit.

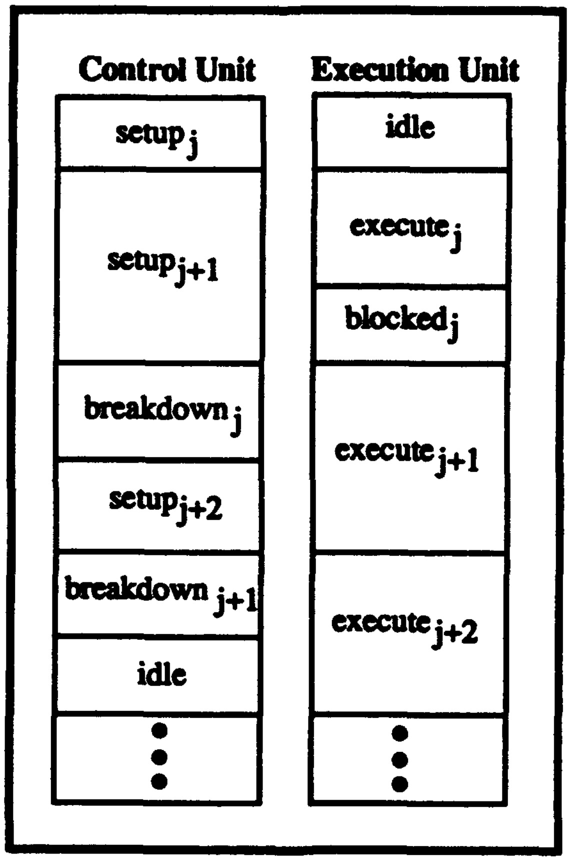

**Figure 2.8. Wrap-Around (Case 2)**

Case 3: both condition (1) and (2) exist. This case is shown in Figure 2.9. For this case wrap-around is possible if:

$$\text{setup}_{j+1} + \text{breakdown}_j + \text{setup}_{j+2} + \text{breakdown}_{j+N} \le \text{execution}_j + \text{execution}_{j+1}$$

Note that if neither condition (1) nor condition (2) applies, wrap-around is not possible.

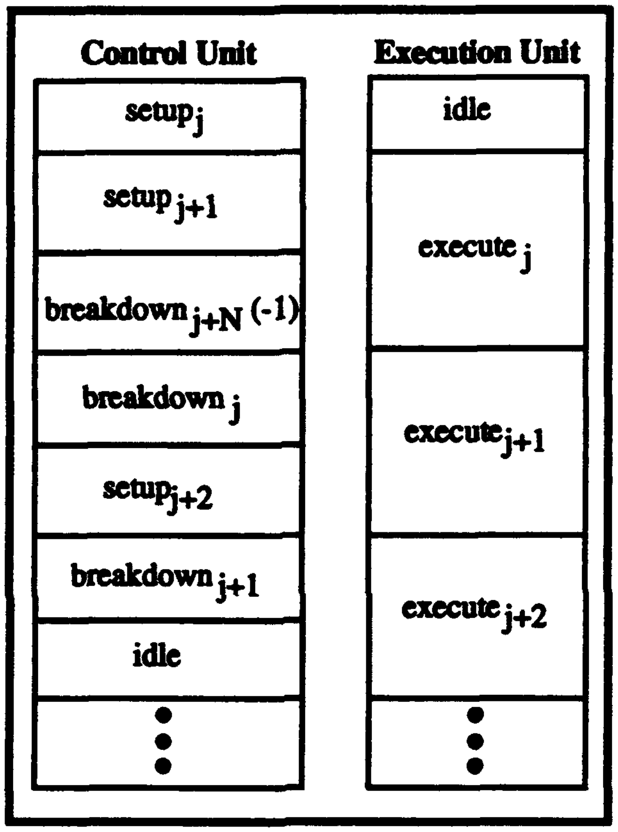

Figure 2.9. Wrap-Around (Case 3)

# III. ALGORITHM FOR NODE ALLOCATION

This chapter discusses the particular node allocation algorithm that addresses the issues discussed in the previous chapter within the concept of the LGDF model. Initial allocation of the nodes to processors is accomplished by the user, taking into account proper load balancing. The remaining issues are handled by the algorithms discussed below.

## A. OVERLAP

Overlap is accomplished by first taking each processor individually and scheduling the node with the greatest execution time first. This algorithm is illustrated in Figure 3.1. The nodes are then scheduled on each processor with regard to overlap as in Figure 3.2.

```
for i = 1 to total_number_of _processors
   for processor $P_i$
      j = 1
      while node j != NULL
         if $execution_j$ <= $execution_{j+1}$
            temp = node j
            node j = node j+1
            j = j+1
         endif
      end while
   end for
end for
```

**Figure 3.1. Execution Cycle Scheduling Algorithm**

In the overlap algorithm, the second node on the processor is initially compared to the first node. If the setup of the second node is less than the execution time of the first node, then overlap can occur and the second node is scheduled after the first node.

```
for i = 1 to total_number_of_processors
    for processor $P_i$
        j = 1
        k = j+1
        schedule
        while node j != NULL
            while execution $_j$ < setup$_k$
                k = k + 1
            end while
            temp = node j+1
            node j+1 = node k
            node k = temp
            j = j+1
        end while
    end for
end for
```

**Figure 3.2. Overlap Algorithm**

If this condition is not true, the following node on the processor is then compared to the first node and this process continues until a suitable node is found or until all nodes on the processor have been checked. If all nodes on the processor have been checked and none are found suitable, or if a node has been found which meets the conditions, the node is scheduled and this node is then compared to the remaining nodes. This process continues until all nodes on the processor have been exhausted. This scheduling method is performed on each processor in turn. It is assumed that since the nodes are initially scheduled in decreasing order of execution that the breakdown of the previous node will likely be less than the execution time of the next node.

## B. WRAP-AROUND

The wrap-around algorithm is shown in Figure 3.3. For each processor, the breakdown time of the last node is taken and summed with the setup time of the second node and the breakdown time of the first node. This sum is compared to the sum of the execution times of the first and second nodes. If the sum of the setup and breakdown times is less than the sum of the execution times, the last node breakdown time can be wrapped-around. There are several other conditions which can also occur. Typically, for more than three nodes scheduled on a processor, it is possible for the setup time of the third node to

```
for i = 1 to total_number_of _processors
   for processor $P_i$
      $j = 1$
      if $\text{breakdown}_j + \text{setup}_{j+1} + \text{breakdown}_{j+N} <= \text{execution}_j + \text{execution}_{j+1}$
         start $\text{breakdown}_{j+N}$ @ $(\text{setup}_j + \text{breakdown}_j + \text{setup}_{j+1})$ cycles
      end if
   end for
end for
```

**Figure 3.3. Wrap-Around Algorithm**

occur during the execution time of the second node after the breakdown of the first node. In this case, the setup time of the third node is also summed with the setup time of the second node and the breakdown times of the first node and the last node.

# IV. RUN-TIME PERFORMANCE

This chapter describes the results for use of the revolving cylinder algorithm. The programs used for generation of the results are fully described in Appendix B. Figure 4.1 is a graphical summary of the programs and their related inputs and outputs.

## A. PERFORMANCE METRICS

The performance evaluations for the RC technique were generated using an actual application graph called a correlator [Ref. 4]. This graph is illustrated in Figure 4.1. The RC technique that was analyzed was the start after finish (SAF) technique. The results from this technique were compared to an FCFS scheduling algorithm. Simulations were performed on cylinders generated for both wrap-around and non wrap-around techniques.

Several initial assumptions were made for the RC cylinders. The scheduler latency, node setup and breakdown latency, and instruction size were assumed to be zero. The read, write, produce, consume, and threshold amounts for an individual queue were assumed to be equal. The queue capacity was calculated as eight times the queue threshold. Nodes were manually allocated to processors based on load balancing and minimizing queue contention; that is, no processor would simultaneously access the same queue for reading and writing. As many memory modules as necessary to completely eliminate memory module contention were then assigned to processors. The number of memory modules required was based on the static representation of the cylinder generated by the scheduler and mapping programs. Eight processors were used in the system and the node to processor allocation was identical throughout the simulations.

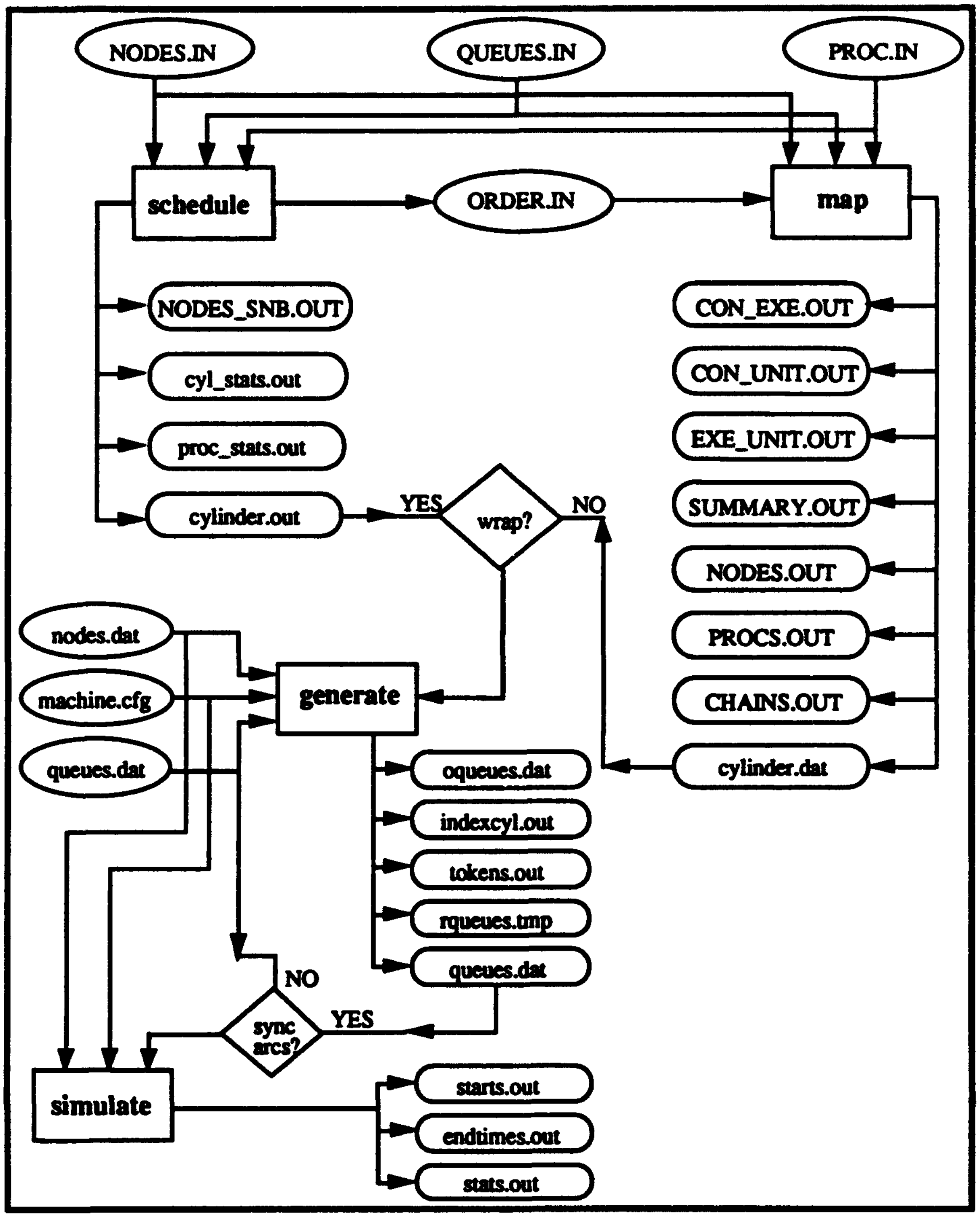

**Figure 4.1. Revolving Cylinder Program Summary**

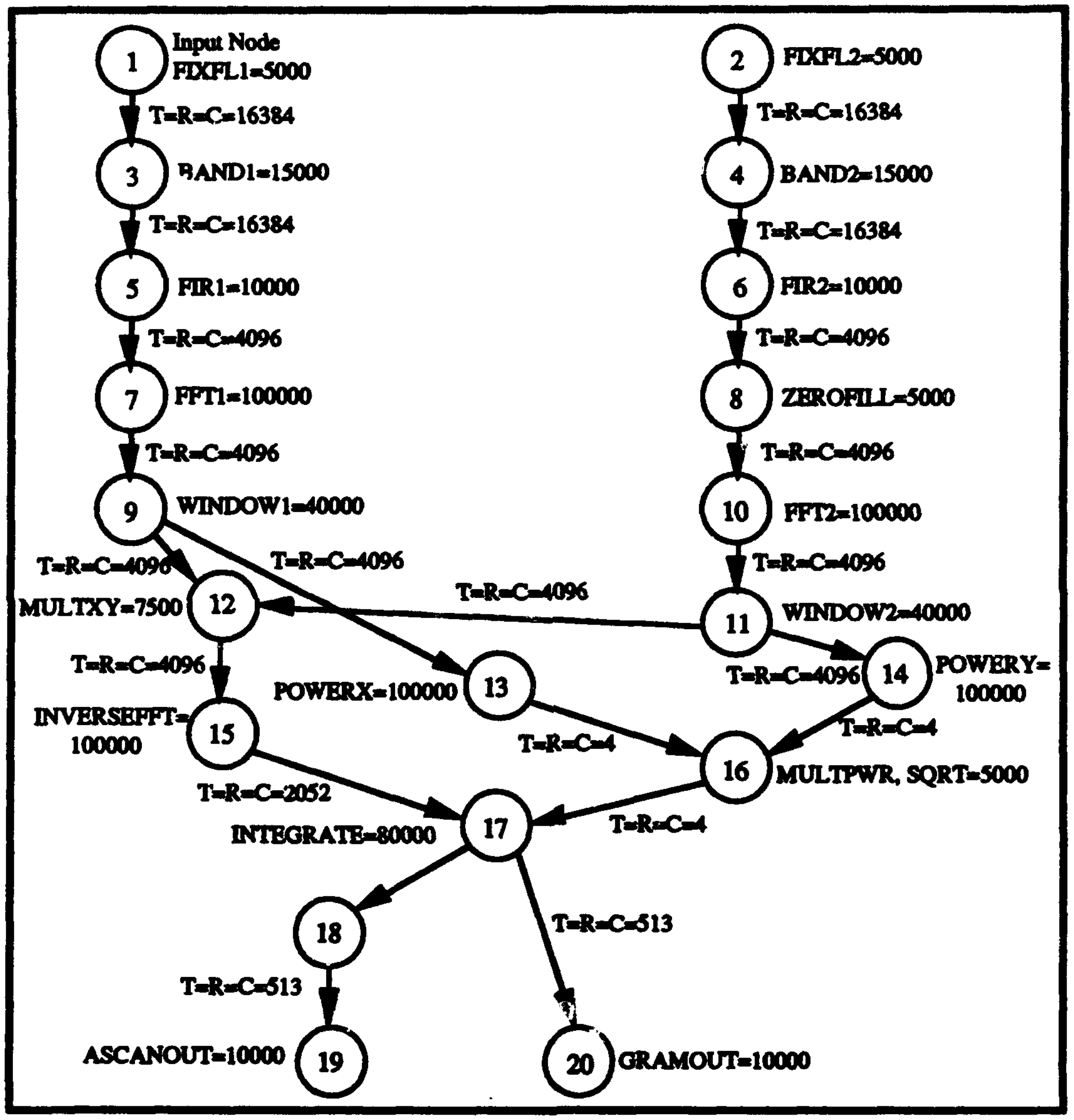

**Figure 4.2. Correlator Graph [Ref. 5]**

## B. RESULTS

Figures 4.3 and 4.4 illustrate the normalized maximum throughput for the correlator versus the ratio of communication cycles to computation cycles. The communication costs used for the mapping were varied from 3 to 23 cycles to transfer one word of data from a processor to memory. These correspond to communication/computation ratios of 0.1 to 0.77, respectively. The theoretical minimum average input period was used as the

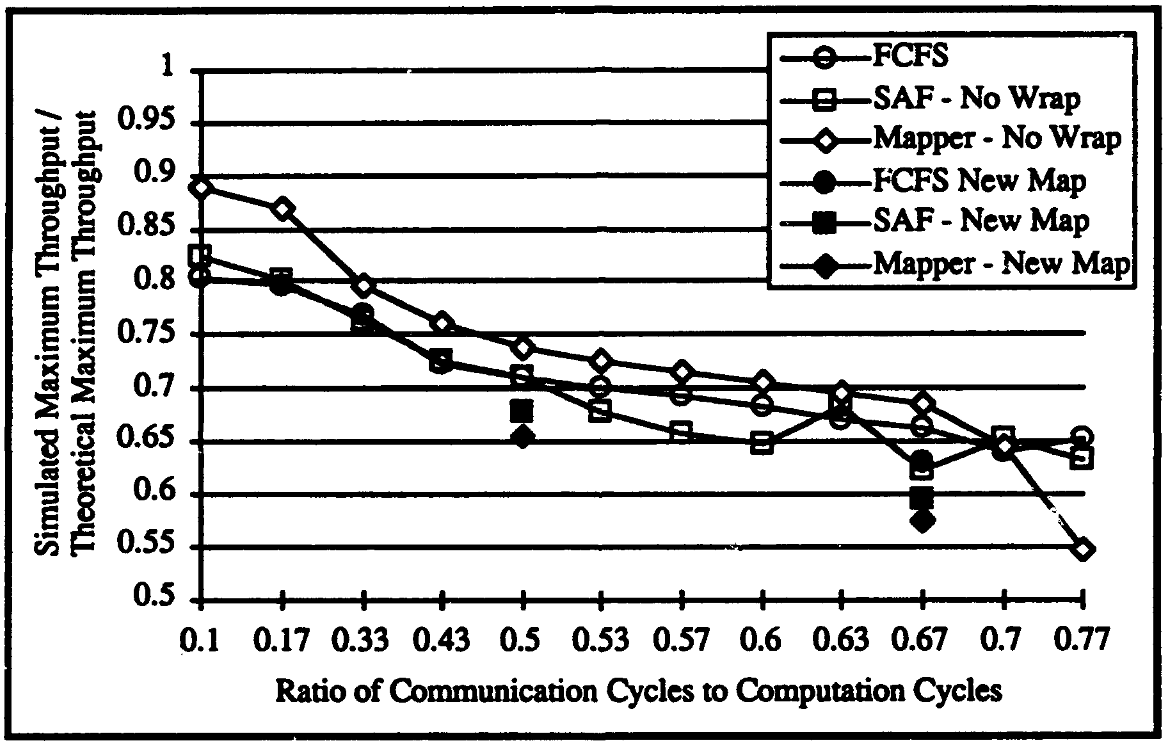

**Figure 4.3. Normalized Maximum Throughput vs. Communication/Computation (No Wrap)**

normalizing factor. This normalizing factor was calculated by taking the inverse of the ideal cylinder calculation for one instance of the graph and multiplying by $1x10^6$. The $1x10^6$ factor is necessary since maximum throughput is given by the simulator in instances per megacycle.

The 'mapper' points listed in the legend represent the maximum theoretical throughput for the compile-time representation of the cylinder. This value is obtained by taking the inverse of the end time of activities obtained by the map program multiplied by $1 \times 10^6$. In Figure 4.4, there are two representations of the mapper. The first is a 'flat' cylinder. Each

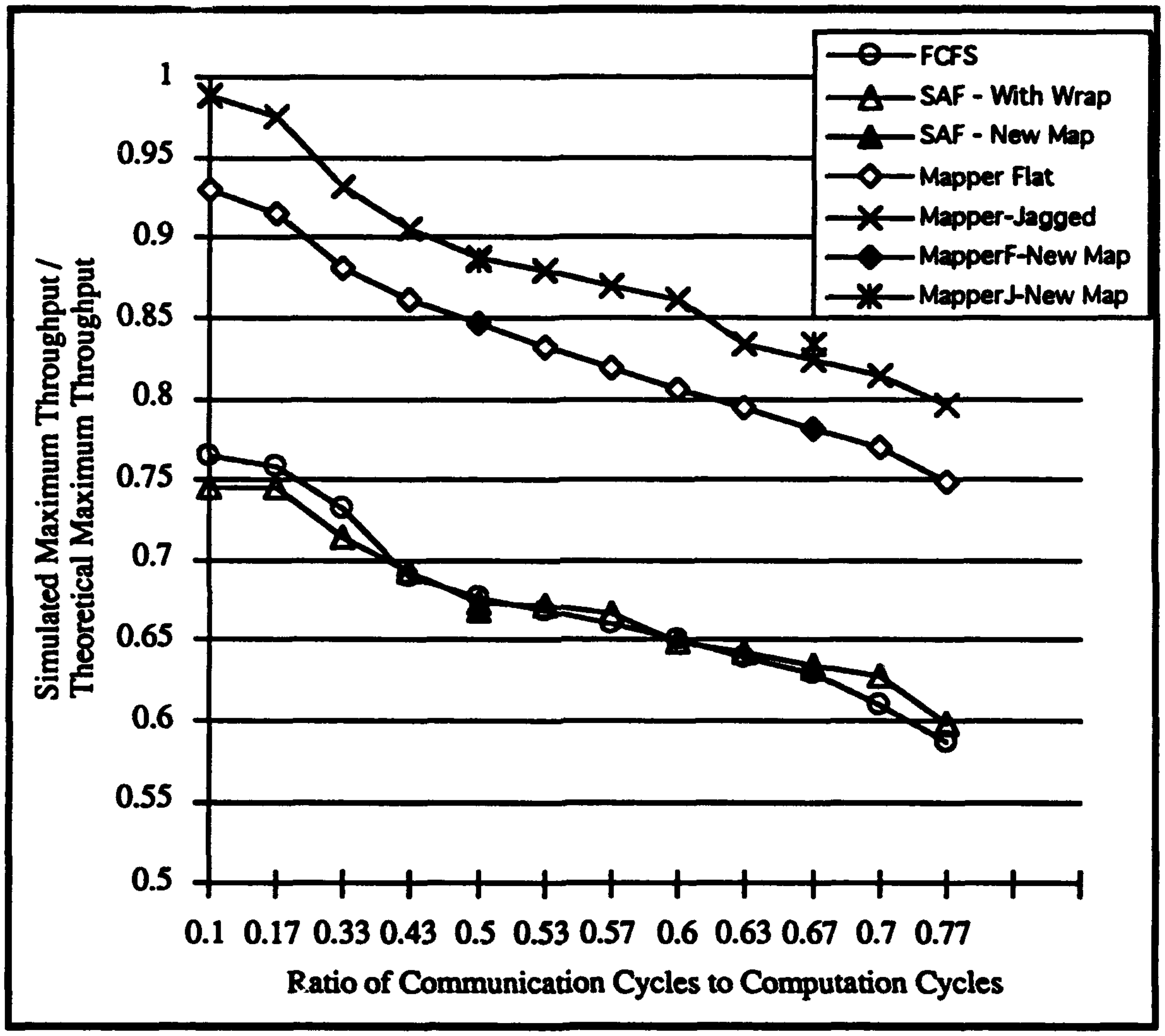

**Figure 4.4. Normalized Maximum Throughput vs. Communication/Computation (With Wrap)**

static cylinder slice of the graph ends at different time, represented as a number of cycles. The 'flat' cylinder takes the greatest end time of all cylinder slices and uses that value as the average end time of the graph. This means if a cylinder slice ends before this average end time, idle cycles may be added to the execution unit, thereby decreasing the calculated throughput. The 'jagged' cylinder, however, takes into account each individual cylinder slice end time, and uses the average of the end times over all cylinder slices as the average end time. Thus, maximum throughput for the 'jagged' cylinder is greater than the 'flat' cylinder.

In both Figure 4.3 and Figure 4.4, note that as communication costs increase, SAF results in better throughput than FCFS. This is due to the ability in SAF to map the nodes to minimize contention [Ref. 2].

Since the node to processor allocation was identical throughout the simulations, it was desirable to see if different allocation at various communication costs would have an effect on throughput. A separate node to processor allocation was tried for 15 and 20 cycles to transfer one word of data from a processor to memory. The allocation of nodes was modified only slightly, i.e., only one node was allocated to a different processor. These points are indicated in Figures 4.3 and 4.4 as 'New Map'. It is clear for both wrap and no wrap cases that the revolving cylinder values (SAF and mapper) are affected by slight changes in the node allocation.

Figures 4.5 and 4.6 represent the normalized response time and the coefficient of variation of normalized response time for both the no wrap and wrap cases, respectively. The normalizing factor used in Figure 4.5 is the number of execution cycles required for the completion of one iteration of the critical path of the graph. Note that the response time for SAF (both no wrap and wrap cases) is lower than FCFS at high communication costs. Note also that although SAF no wrap has a slightly better response time than with wrap at high communication costs, modifying the node to processor allocation (New Map) has a significant affect on the no wrap case. Thus, it is possible to improve the response times for both cases by changing the node allocation.

The coefficient of variation represented in Figure 4.6 is a measured comparison between the response times of all graph instances to the average response time. The lower this number, the closer the measured response times are to the average [Ref. 2]. SAF with wrap appears to have the best overall performance as measured by coefficient of variation throughout the range of communication costs. Again, however, modification of the node to processor allocation significantly affects the results, thus indicating that coefficient of variation could be improved over FCFS for both SAF cases.

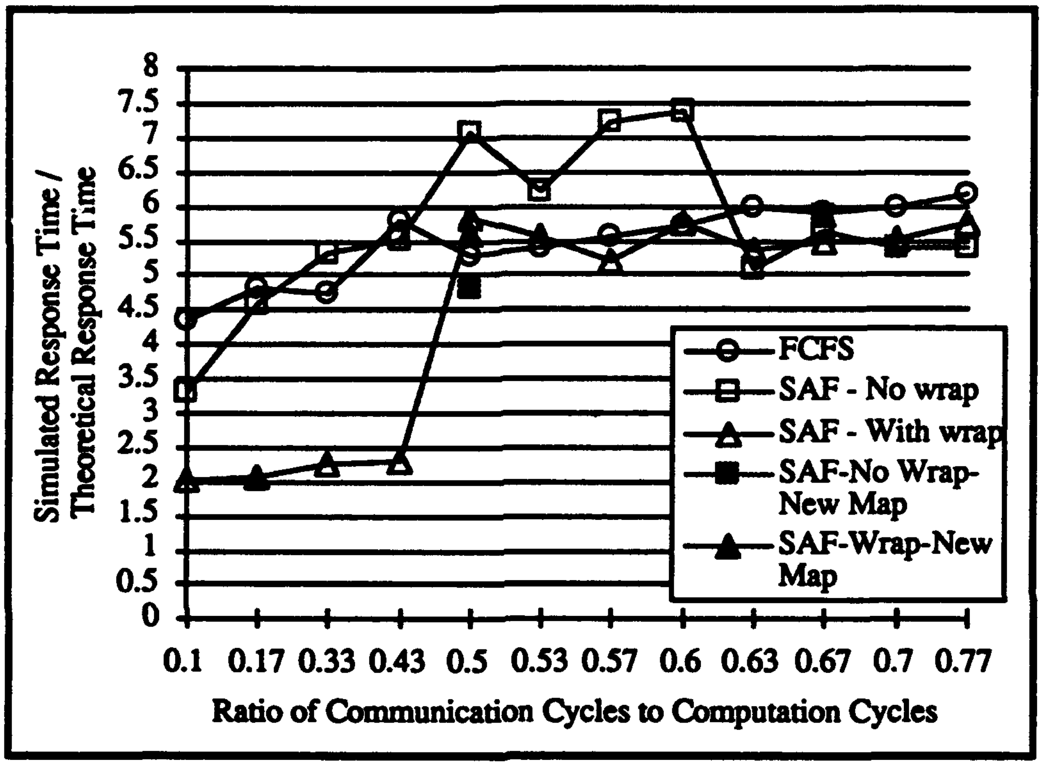

**Figure 4.5. Normalized Response Time vs. Communication/Computation**

Figures 4.7, 4.8 and 4.9 represent normalized maximum throughput for communication costs of 3 cycles, 5 cycles, and 15 cycles versus load. Load in this case is based on fractional multiples of the maximum throughput case (1.0 in the figure). These multiples correspond to a range of graph input from severe lack of input data to overflow of data. From these figures, SAF results in slightly better overall throughput at higher graph loads versus FCFS. Although SAF no wrap performs better than SAF with wrap at low communication costs, SAF with wrap achieves a higher overall throughput over SAF no wrap and FCFS at high communication costs for the entire range of loads, which is the desired result.

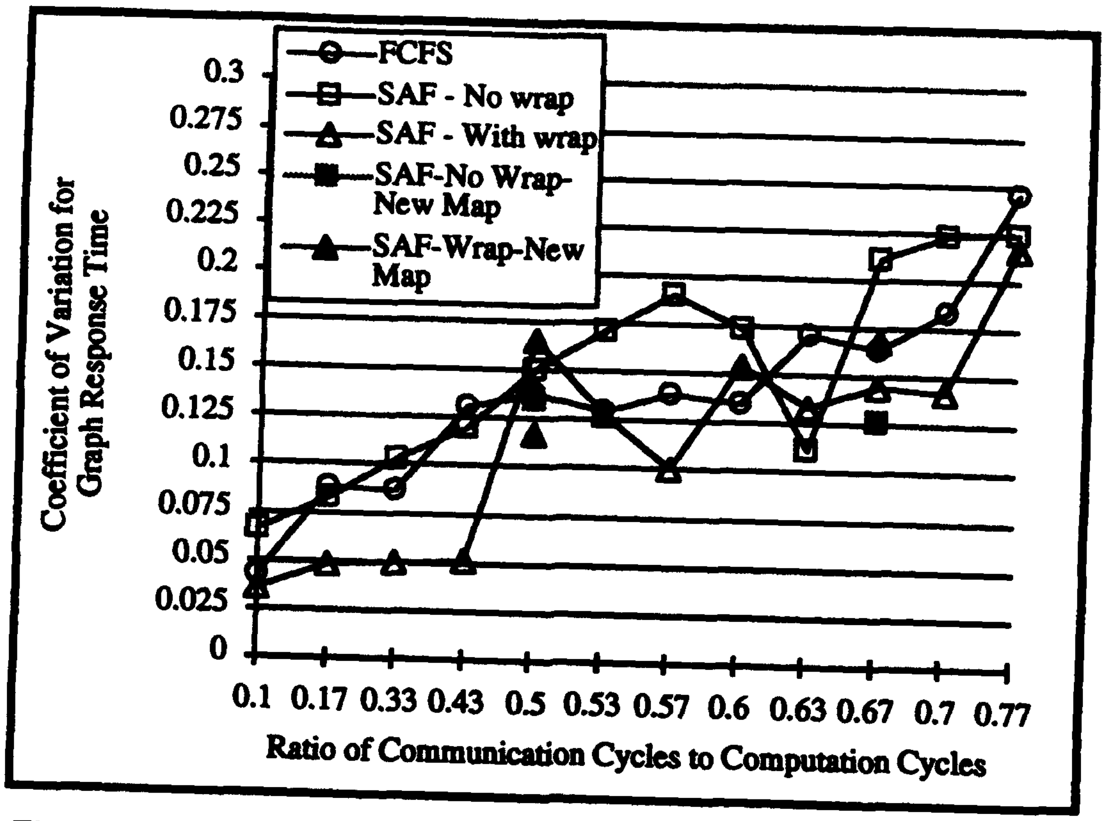

**Figure 4.6. Coefficient of Variation vs. Communication/Computation**

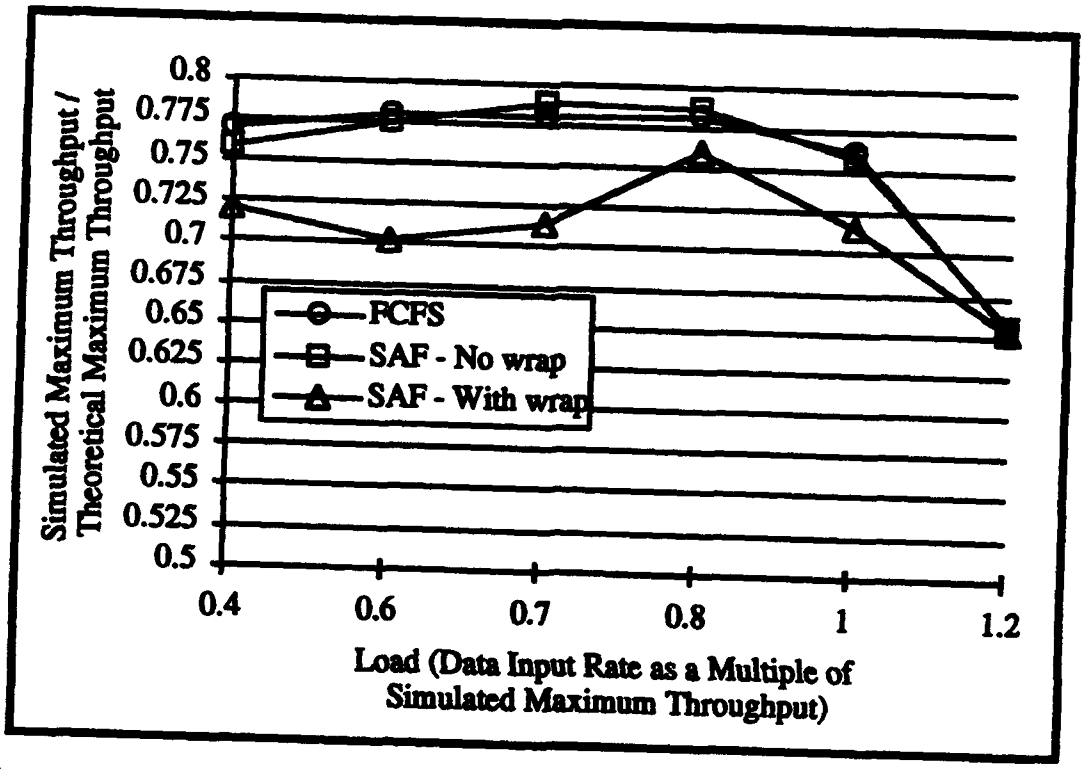

**Figure 4.7. Normalized Maximum Throughput vs. Load (3 Cycles/Word) (0.10 Communication/Computation)**

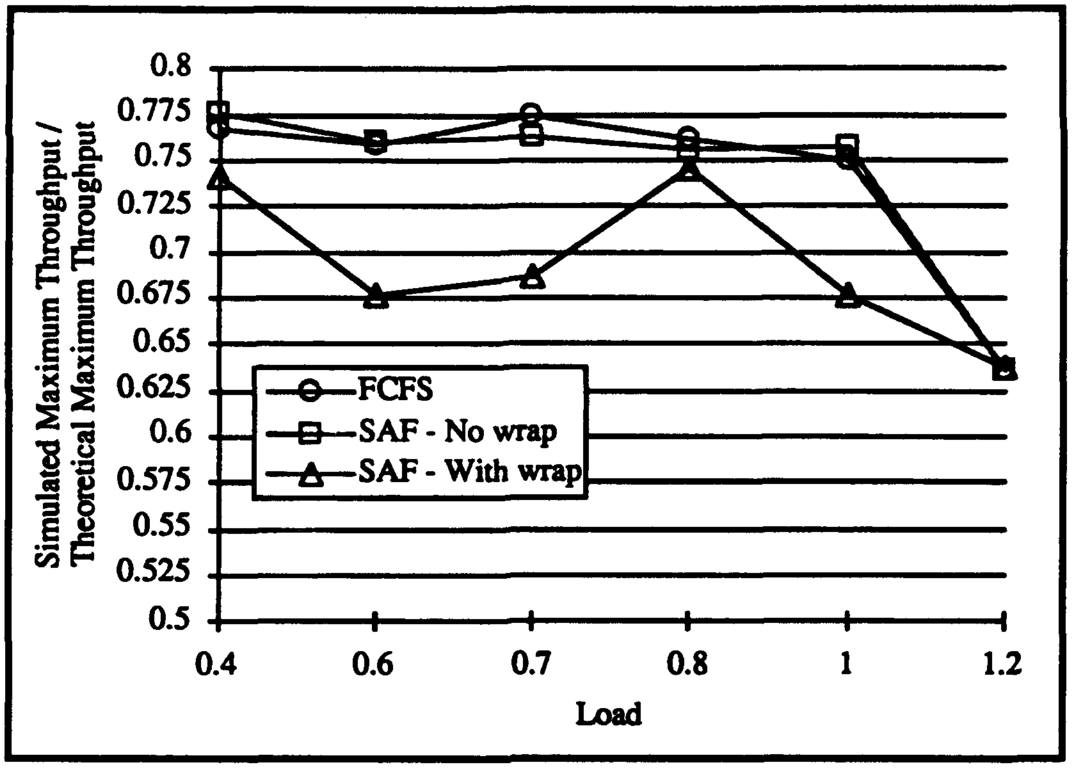

**Figure 4.8. Normalized Maximum Throughput vs. Load (5 Cycles/Word) (0.17 Communication/Computation)**

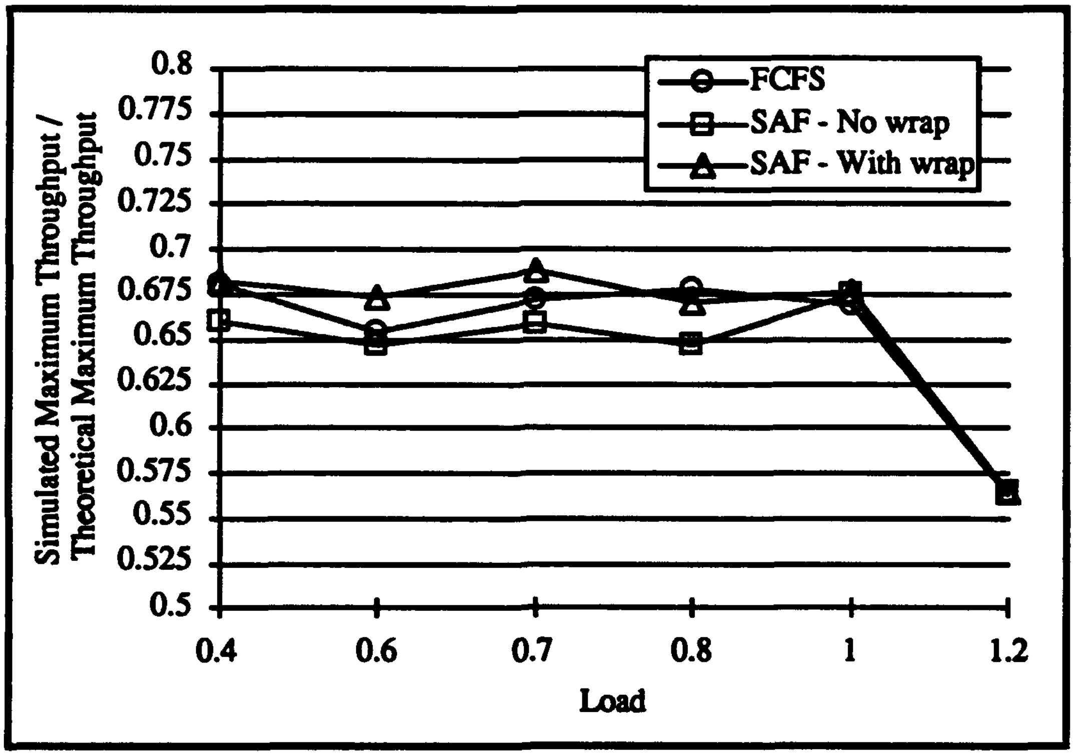

**Figure 4.9. Normalized Maximum Throughput vs. Load (15 Cycles/Word) (0.50 Communication/Computation)**

Figures 4.10, through 4.15 illustrate normalized response time versus load and coefficient of variation versus load for the same communication costs as the three previous figures. From these figures, SAF is shown to have the best overall response time and lowest coefficient of variation throughout the range of load.

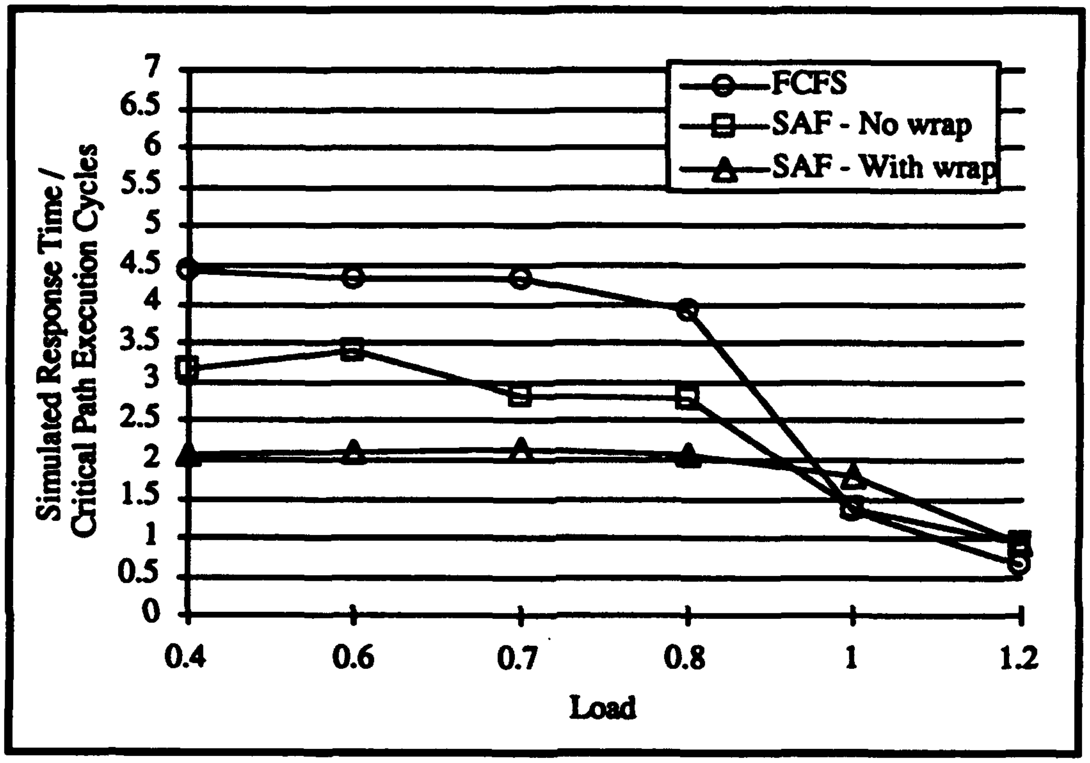

**Figure 4.10. Normalized Response Time vs. Load (3 Cycles/Word) (0.10 Communication/Computation)**

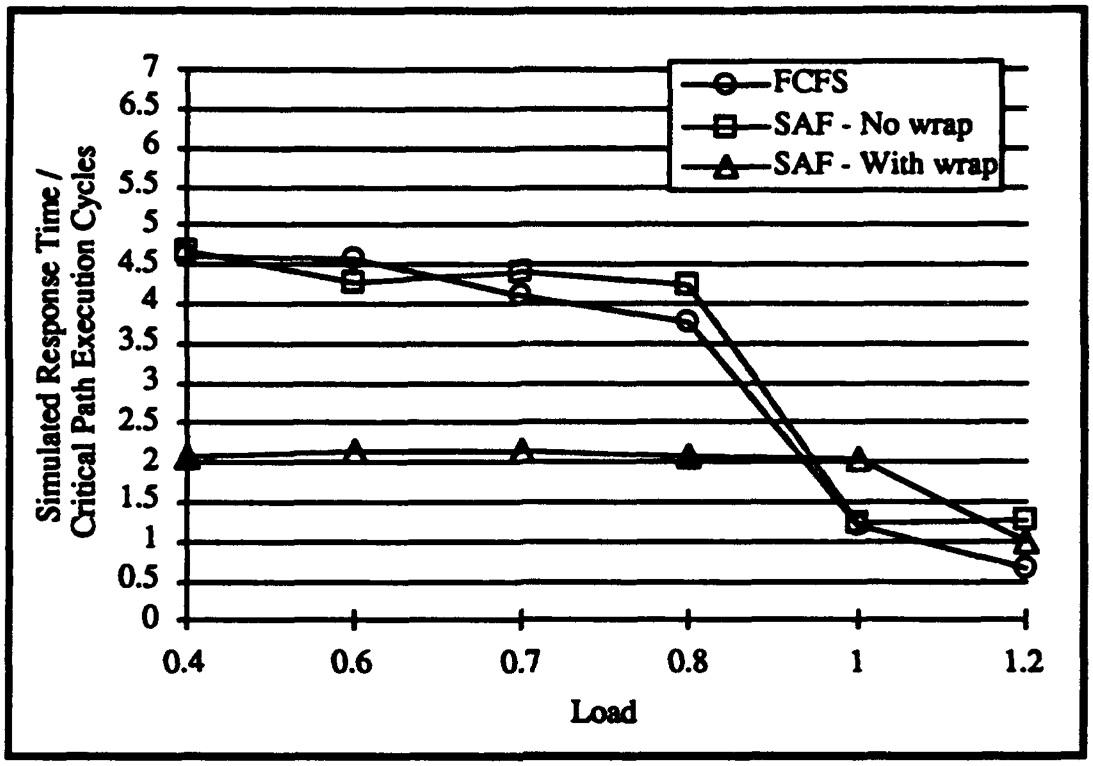

**Figure 4.11. Normalized Response Time vs. Load (5 Cycles/Word) (0.17 Communication/Computation)**

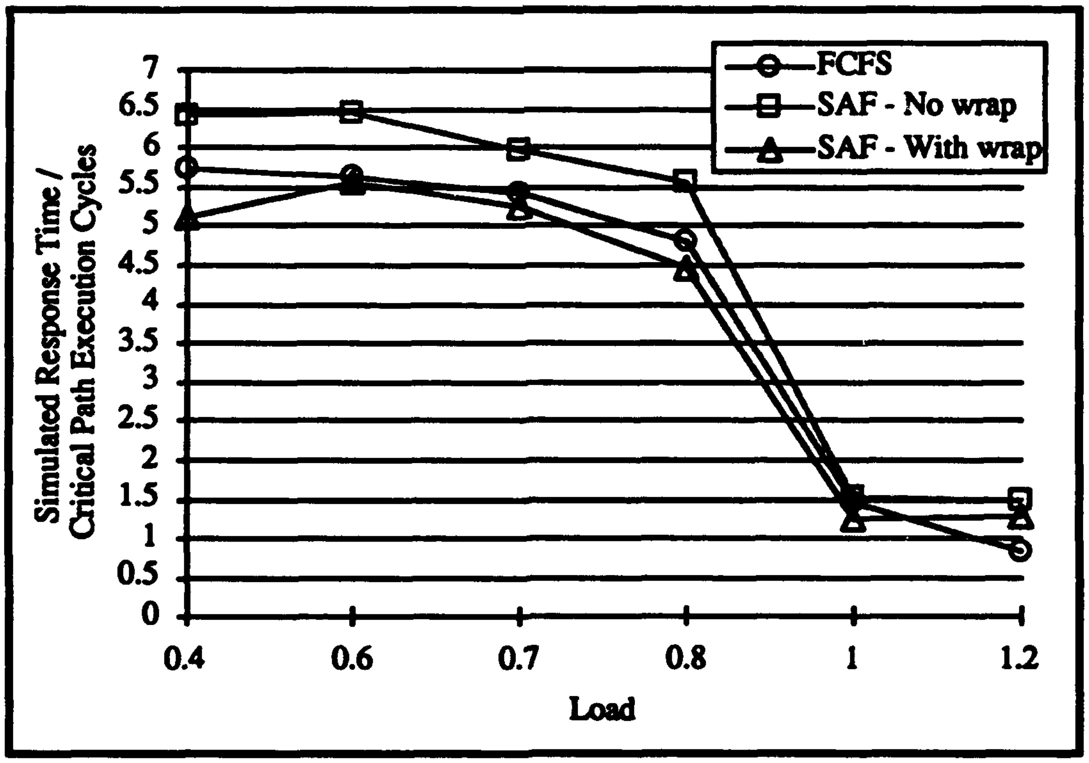

**Figure 4.12. Normalized Response Time vs. Load (15 cycles/Word) (0.5 Communication/Computation)**

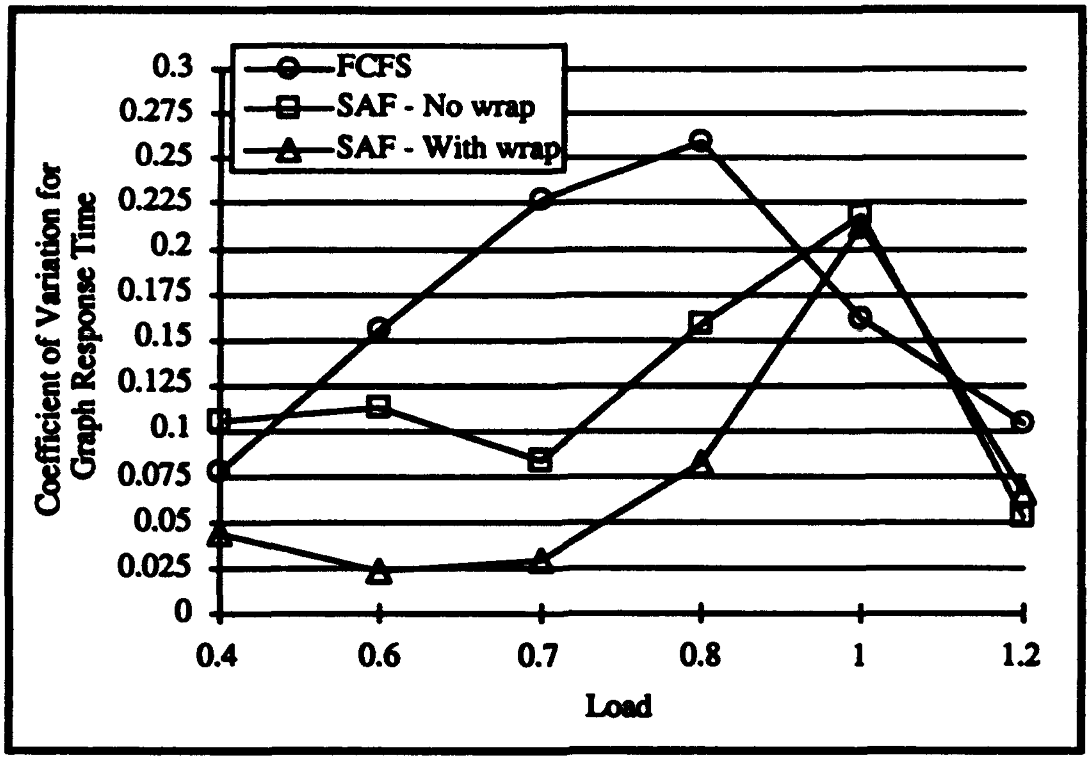

**Figure 4.13. Coefficient of Variation vs. Load (3 Cycles/Word) (0.10 Communication/Computation)**

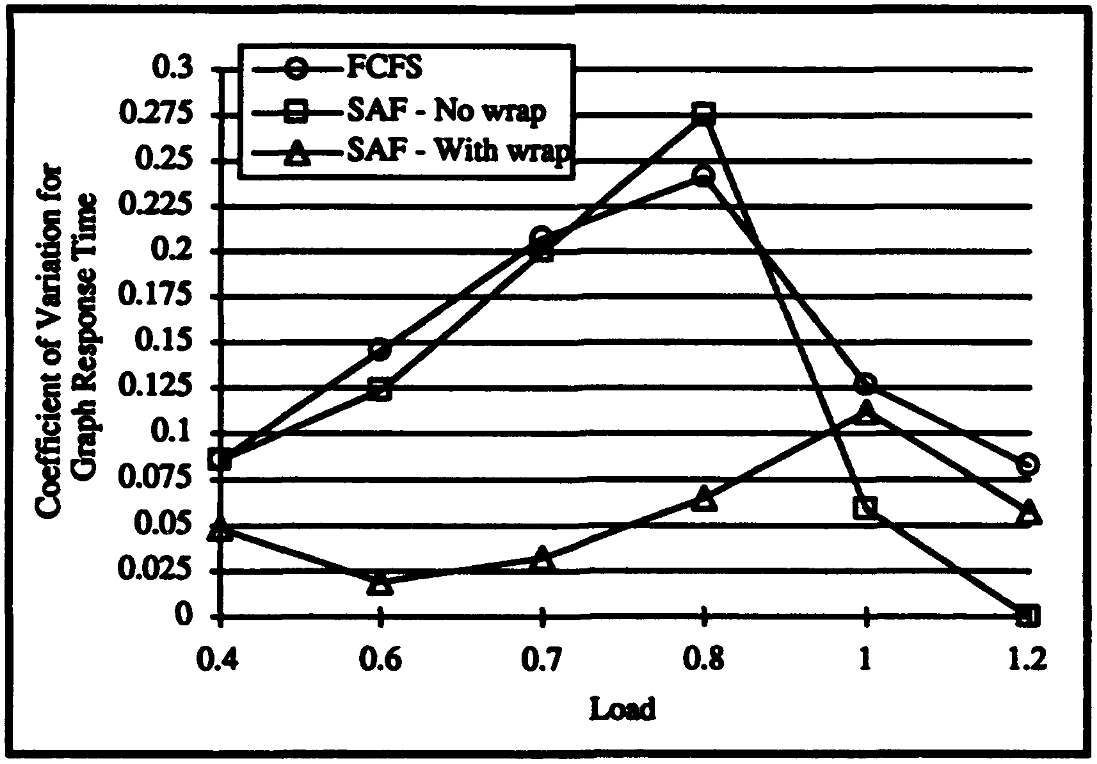

**Figure 4.14. Coefficient of Variation vs. Load (5 Cycles/Word) (0.17 Communication/Computation)**

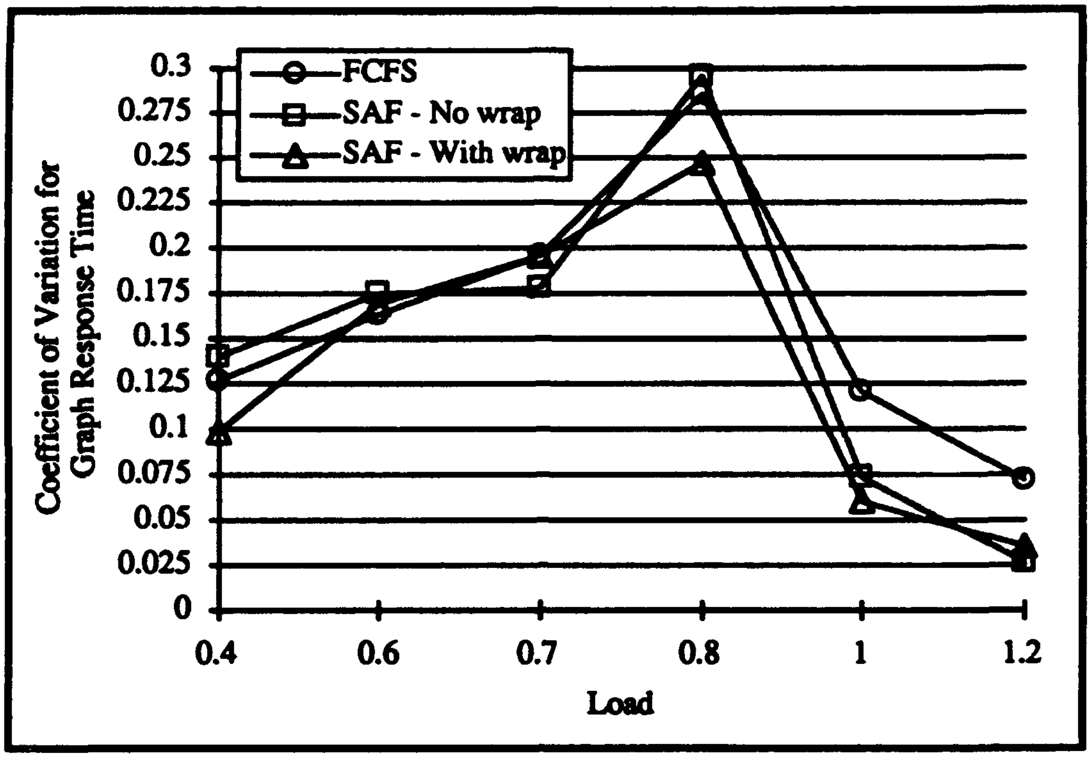

**Figure 4.15. Coefficient of Variation vs. Load (15 Cycles/Word) (0.5 Communication/Computation)**

# V. CONCLUSION

This thesis described the issues involved in node allocation and described a program implemented to resolve those issues. An addition to the RC technique, wrap-around was also analyzed as an improvement to the compile-time implementation of the graph.

A revolving cylinder technique, start-after-finish, was studied and compared to the First-Come-First-Served technique for a large grain data flow graph model. It was demonstrated that RC provides overall better throughput than FCFS, particularly at high communication costs. In addition, it was shown that the RC technique is sensitive to cylinder mapping, especially at high communication costs. Thus, it is important in the analysis of the RC technique to optimize the mapping for each instance of communication cost.

## A. FURTHER RESEARCH

There were several initial assumptions that were made for the graph model that could be removed for future work.

1. The number of instructions for each node was assumed to be zero. Analysis should be conducted with variable instruction lengths.

2. Scheduler latency was also assumed to be zero. This quantity should also be varied and its effect on the RC technique studied.

3. Since the RC results were sensitive to cylinder mapping, it would be desirable to find an optimum cylinder mapping for each level of communication cost. From this a heuristic could be developed such that an extra program module could be added to the existing programs to perform this task automatically.

# APPENDIX A. NODE ALLOCATION PROGRAM

```
// LIEUTENANT JOHN P. CARDANY, U.S. NAVY
// 20 APRIL 1994
// NAVAL POSTGRADUATE SCHOOL
// ADVISORS: PROFESSORS SHRIDHAR SHUKLA AND AMR ZAKY

// Large Grain Data Flow Node to Processor Schedule Program
// schedule.C


#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>


#include "node_alloc.h"

node_alloc cylinder;    //define a cylinder as a node alloc type

int
main()
{
  cout << "\n\nLARGE GRAIN DATA FLOW NODE TO PROCESSOR SCHEDULING
PROGRAM\n\n";

  cout << "\nALLOCATING NODES...\n";

  //System calls
  cylinder.define_times();
  cylinder.read_processor_file();
  cylinder.read_queue_file();
  cylinder.change_node_file();
  cylinder.order_nodes();
  cylinder.sequence nodes();

cout << "\nEND OF PROGRAM\n";
return 0;
}
```

```
// LIEUTENANT JOHN CARDANY, U.S. NAVY
// 20 APRIL 1994
// NAVAL POSTGRADUATE SCHOOL
// ADVISORS:  PROFESSORS SHRIDHAR SHUKLA AND AMR ZAKY

// Node Allocation Class Header File
// node_alloc.h


#ifndef NODE_ALLOC_H
#define NODE_ALLOC_H

#include <fstream.h>
#include <iostream.h>
#define newln '\n'

class node_alloc
{

private:

  // Structure to define queues
  struct queue_type
  {
    int queue_id;
    int source_node;
    int sink_node;
    long write_amount;
    long read_amount;
  };

  // Structure to define nodes
  struct node_type
  {
    int nodes_per_processor;
    int node_id;
    long instr_size;
    long setup_time;
    long exe_time;
    long breakdown_time;
    int proc_type;
    long start_time;
    long end_time;
  };

  // Structure to define ORDER.IN elements
  struct order_in_type
  {
    int node_id;
    long start_time;
  };


  // User inputs to define system
```

```
    int number_of_nodes;              //! Number of Nodes in the System
    int number_of_queues;             // Number of Queues in the System
    int number_of_processors;          // Number of Processors in the System
    long latency;                     // Fixed Scheduler Latency
    long fixed_setup;                 // Fixed Setup time (a)
    int comm;                         // Communication Time for One Word of Information
    long setup, breakdown;            //Setup and Breakdown Times Per Node
    queue_type queue[ 250 ];          // System Queues
    node_type  node[250];             // System Nodes
    node_type  order[50][50];         //Matrix to store Node structures
    node_type  new_order[50][50];     //Matrix to store ordered Node Structures
    order_in_type ORDER[250];         //Node order matrix

  public:

    // Class Constructor
    node_alloc();

    // Function to load timing information into the system
    void define_times();

    // Function to read number of processors into the system
    void read_processor_file();

    // Function to load queue data into the system
    void read_queue_file();

    // Function to load the node data into the system
    void change_node_file();

    // Function to Order the Nodes and Create the ORDER.IN File
    void order_nodes();

    // Function to calculate the unused execution cycles
    void calc_unused_exe_cycles();

    // Function to implement wrap-around
    void wrap_around();
    //Function to Create cylinder file
    void make_cylinder_file(long);

    //Function to print processor statistics
  void generate_processor_stats(int,long,long,long,long,long,long, long);

    //Function to print cylinder statistics
  void generate_statistics(long,long,long,long,long,long,long,long);
  // Function to reorder the ORDER.IN file sequentially
void sequence_nodes();

  //Class Destructor
  ~node_alloc() {}
};
#endif
```

```
// LIEUTENANT JOHN P. CARDANY, U.S.NAVY
// 28 April 1994
// NAVAL POSTGRADUATE SCHOOL
// ADVISORS:  PROFESSORS SHRIDHAR SHUKLA AND AMR ZAKY

// Input-Output Data Class Source File
// node_alloc.C


#include "node_alloc.h"

#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>


// Class Constructor
node_alloc::node_alloc()
{
  number_of_nodes = 0;
  number_of_queues = 0;
  number_of_processors = 0;
  comm = 0;
}


// Function to Load Timing Information into the System
void
node_alloc::define_times()
{
  cout << "\nFixed Setup Time (cycles)   : ";
  cin >> fixed_setup;
  cout << "\nWord Communications Time (cycles)  : ";
  cin >> comm;
  if ( fixed_setup < 0 || comm < 0 )
  {
    cerr << "\nInvalid Communication Time\n";
    exit(0);
  }
}


// Function to Read the Processor File
void
node_alloc::read_processor_file()
{
 ifstream processor_input_file;
 processor_input_file.open("PROCS.IN");
 if ( !processor_input_file )
  {
   cerr << "\nCannot Open file PROCS.IN\n";
   exit(0);
  }
 processor_input_file >> number_of_processors;
```

```
  cout << "\nNumber of Processors: " << number_of_processors << "\n\n";
  processor_input_file.close();
 }

// Function to Load the Queue Data File
void
node_alloc::read_queue_file()
{
  ifstream queue_input_file;
  queue_input_file.open( "QUEUES.IN" );
  if ( !queue_input_file )
  {
    cerr << "\nCannot open file QUEUES.IN\n";
    exit(0);
  }
  queue_input_file >> number_of_queues;
  for ( int cnt = 0; cnt < number_of_queues; cnt++ )
  {
    queue_input_file >> queue[ cnt ].queue_id;
    queue_input_file >> queue[ cnt ].source_node;
    queue_input_file >> queue[ cnt ].sink_node;
    queue_input_file >> queue[ cnt ].write_amount;
    queue_input_file >> queue[ cnt ].read_amount;
    if ( queue[ cnt ].queue_id  <= 0 )
    {
         cerr << "\nInvalid Queue ID or Wrong Quantity\n";
         exit(0);
    }
    if ( queue[ cnt ].write_amount < 0 || queue[ cnt ].read_amount < 0 )
    {
         cerr << "\nInvalid Parameter for Queue : " << setw(6);
         cerr << queue[ cnt ].queue_id << endl;
         exit(0);
    }
    for ( int cntq = 0; cntq < cnt; cntq++ )
    {
         if ( queue[ cntq ].queue_id == queue[ cnt ].queue_id )
         {
           cerr << "\nDuplicated Queue ID : " << setw(6);
           cerr << queue[ cnt ].queue_id << endl;
           exit(0);
         }
    }
  }
  queue_input_file.close();
}


// Function to Change the Node Data File
void
node_alloc::change_node_file()
{
  ifstream node_input_file;
  node_input_file.open( "NODES.IN" );
  if ( !node_input_file )
  {
    cerr << "\nCannot open file NODES.IN\n";
    exit(0);
  }
  ofstream node_output_file;
  node_output_file.open("TEMP.OUT");
  int n_id, i_size, p_type;
  unsigned long s_time, e_time, b_time;        //s=setup, b=breakdown, e=execution
  node_input_file >> number_of_nodes;
  node_output_file << number_of_nodes << newln << newln;
  for ( int cnt = 0; cnt < number_of_nodes; cnt++ )
  {
    node_input_file >> n_id >> i_size >> s_time >> e_time >> b_time >> p_type;

    if ( n_id <= 0 )
    {
        cerr << "\nInvalid Node ID or Wrong Quantity\n";
        exit(0);
    }
    if ( s_time < 0 || e_time < 0 || b_time < 0 || i_size < 0 )
    {
        cerr << "\nInvalid Parameter for Node : " << setw(6) << n_id << endl;
        exit(0);
    }

    long setup = 0;
    long breakdown = 0;
    for ( int cntq = 0; cntq < number_of_queues; cntq++ )
    {
        if ( queue[ cntq ].source_node == n_id )
        {
          breakdown += ( comm * queue[ cntq ].write_amount );
        }
        if ( queue[ cntq ].sink_node == n_id )
        {
          setup += ( comm * queue[ cntq ].read_amount );
        }
    }
    setup += fixed_setup;
    breakdown += fixed_setup;
    s_time = setup;
    b_time = breakdown;

node_output_file << n_id << setw(4) << i_size << setw(8) << s_time << setw(12) <<
e_time << setw(13)<< b_time << setw(14) << p_type << newln;

  }

  node_input_file.close();
  node_output_file.close();
  system("mv NODES.IN NODES_IN.ORG");
  system("mv TEMP.OUT NODES.IN");
}
```

```
// Function to Schedule the Nodes and create the ORDER.IN file
void
node_alloc::order_nodes()
{
  int loop, count, tail_index, swap_index;
  int SWAPP[illegible] T_MOVE;
  node_type [illegible]d_ptr, *Curr_ptr, *Tail_ptr;
  node_type TEMP_NODE;

  ifstream node_input_file;
  node_input_file.open("NODES.IN");
  ofstream order_output_file;
  order_output_file.open("ORDER.IN");
  node_input_file >> number_of_nodes;

  for ( int cnt = 0; cnt < number_of_nodes; cnt++ )   //Place nodes in an array
  {
     node[cnt].nodes_per_processor = 0;
     node_input_file >> node[cnt].node_id;
     node_input_file >> node[cnt].instr_size;
     node_input_file >> node[cnt].setup_time;
     node_input_file >> node[cnt].exe_time;
     node_input_file >> node[cnt].breakdown_time;
     node_input_file >> node[cnt].proc_type;
     node[cnt].start_time = 0;

  }

  for (int i = 0; i < number_of_processors; i++ )       //Place the nodes in two 2x2 matrices
  {
    count = 0;

      for (int j = 0; j < number_of_nodes; j++ )
      {
          if ( node[j].proc_type == i+1 )
          {
                order[count][i] = node[j];
                order[count][i].nodes_per_processor = count+1;
                new_order[count][i] = node[j];
                new_order[count][i].nodes_per_processor = count+1;
                count++;
                order[count][i].node_id  = NULL;
                new_order[count][j].node_id = NULL;
          }
      }
  }

        // Order Nodes in decreasing Exe time

 for (int j = 0; j < number_of_processors; j++ )
 {
    int node_index = 0;
    SWAPPED = 1;
    while ( SWAPPED )
    {
         SWAPPED = 0;
         for ( i = 0; i < number_of_nodes; i++ )
         {
           if ( order[i][j].exe_time < order[i+1][j].exe_time )          //then swap nodes
           {
                 TEMP_NODE = order[i][j];
                 order[i][j] = order[i+1][j];
                 order[i+1][j] = TEMP_NODE;
                 SWAPPED = 1;                              //and set a flag
           }
         }
    }
  }


  // Order nodes by comparing Exe and Setup times

  for (j = 0; j < number_of_processors; j++ )
  {
   int node_index = 0;
   T_MOVE = 0;

   Head_ptr = &order[node_index][j];
   Tail_ptr = &order[node_index+1][j];
   Curr_ptr = Tail_ptr;

          if ( Tail_ptr->node_id == NULL ) // only one node
          {
            order_output_file << Head_ptr->node_id << setw(8) << Head_ptr->start_time
<< newln;
          }

   while ( Tail_ptr->node_id != NULL )       //check all nodes on a processor
   {
   SWAPPED = 0;
   T_MOVE = 0;

   swap_index = node_index;

       while ( Head_ptr->exe_time < Tail_ptr->setup_time ) //keep swapping nodes until
       {                                                   // condition fails
           tail_index = swap_index + 2;
           Tail_ptr = &order[tail_index][j];    // point to next node to check condition again
           T_MOVE = 1;        //set flag to indicate tail ptr was moved
           swap_index++;
       }

      // swap Tail_ptr and Curr_ptr to put tail node in position after head node

        if ( T_MOVE && Tail_ptr->node_id != NULL )
```

```
                {
                TEMP_NODE.node_id = Curr_ptr->node_id;
                TEMP_NODE.instr_size = Curr_ptr->instr_size;
                TEMP_NODE.setup_time = Curr_ptr->setup_time;
                TEMP_NODE.exe_time = Curr_ptr->exe_time;
                TEMP_NODE.breakdown_time = Curr_ptr->breakdown_time;
                TEMP_NODE.proc_type = Curr_ptr->proc_type;
                TEMP_NODE.start_time = Curr_ptr->start_time;

                Curr_ptr->node_id = Tail_ptr->node_id;
                Curr_ptr->instr_size = Tail_ptr->instr_size;
                Curr_ptr->setup_time = Tail_ptr->setup_time;
                Curr_ptr->exe_time = Tail_ptr->exe_time;
                Curr_ptr->breakdown_time = Tail_ptr->breakdown_time;
                Curr_ptr->proc_type = Tail_ptr->proc_type;
                Curr_ptr->start_time = Tail_ptr->start_time;

                Tail_ptr->node_id = TEMP_NODE.node_id;
                Tail_ptr->instr_size = TEMP_NODE.instr_size;
                Tail_ptr->setup_time = TEMP_NODE.setup_time;
                Tail_ptr->exe_time = TEMP_NODE.exe_time;
                Tail_ptr->breakdown_time = TEMP_NODE.breakdown_time;
                Tail_ptr->proc_type = TEMP_NODE.proc_type;
                Tail_ptr->start_time = TEMP_NODE.start_time;

                SWAPPED = 1;      //set flag to indicate nodes swapped

                Curr_ptr->start_time = (Head_ptr->setup_time + Head_ptr->start_time);

              // Nodes were not swapped, only two nodes in array
                    if ( node_index == 0 )          //node is head ptr, put node in new array
                    {
                     order_output_file << Head_ptr->node_id << setw(8) << Head_ptr-
>start_time << newln;
                     new_order[node_index][j].node_id = Head_ptr->node_id;
                     new_order[node_index][j].instr_size = Head_ptr->instr_size;
                     new_order[node_index][j].setup_time = Head_ptr->setup_time;
                     new_order[node_index][j].exe_time = Head_ptr->exe_time;
                     new_order[node_index][j].breakdown_time = Head_ptr-
>breakdown_time;
                     new_order[node_index][j].proc_type = Head_ptr->proc_type;
                     new_order[node_index][j].start_time = Head_ptr->start_time;

                    }
                          //Put node after head node into order file and array
                order_output_file << Curr_ptr->node_id << setw(8) << Curr_ptr->start_time <<
newln;
                new_order[node_index+1][j].node_id = Curr_ptr->node_id;
                new_order[node_index+1][j].instr_size = Curr_ptr->instr_size;
                new_order[node_index+1][j].setup_time = Curr_ptr->setup_time;
                new_order[node_index+1][j].exe_time = Curr_ptr->exe_time;
                new_order[node_index+1][j].breakdown_time = Curr_ptr->breakdown_time;
                new_order[node_index+1][j].proc_type = Curr_ptr->proc_type;
                new_order[node_index+1][j].start_time = Curr_ptr->start_time;
```

```
        }

    if ( Tail_ptr->node_id != NULL && !SWAPPED )  //Nodes were not swapped,
    {                                             //no node matches requirement
                                                  //so sked node after head node
    Tail_ptr->start_time = (Head_ptr->setup_time + Head_ptr->start_time);

            if ( node_index == 0 )
            {
             order_output_file << Head_ptr->node_id << setw(8) << Head_ptr-
>start_time << newln;
             new_order[node_index][j].node_id = Head_ptr->node_id;
             new_order[node_index][j].instr_size = Head_ptr->instr_size;
             new_order[node_index][j].setup_time = Head_ptr->setup_time;
             new_order[node_index][j].exe_time = Head_ptr->exe_time;
             new_order[node_index][j].breakdown_time = Head_ptr-
>breakdown_time;
             new_order[node_index][j].proc_type = Head_ptr->proc_type;
             new_order[node_index][j].start_time = Head_ptr->start_time;
            }
     order_output_file << Tail_ptr->node_id << setw(8) << Tail_ptr->start_time <<
newln;
     new_order[node_index+1][j].node_id = Tail_ptr->node_id;
     new_order[node_index+1][j].instr_size = Tail_ptr->instr_size;
     new_order[node_index+1][j].setup_time = Tail_ptr->setup_time;
     new_order[node_index+1][j].exe_time = Tail_ptr->exe_time;
     new_order[node_index+1][j].breakdown_time = Tail_ptr->breakdown_time;
     new_order[node_index+1][j].proc_type = Tail_ptr->proc_type;
     new_order[node_index+1][j].start_time = Tail_ptr->start_time;
    }
    else    // last node to be scheduled
    {
       if ( Curr_ptr->node_id != NULL && !SWAPPED) //sked last node in the array
       {
          Curr_ptr->start_time = (Head_ptr->setup_time + Head_ptr->start_time);
          order_output_file << Curr_ptr->node_id << setw(8) << Curr_ptr-
>start_time << newln;
          new_order[node_index+1][j].node_id = Curr_ptr->node_id;
          new_order[node_index+1][j].instr_size = Curr_ptr->instr_size;
          new_order[node_index+1][j].setup_time = Curr_ptr->setup_time;
          new_order[node_index+1][j].exe_time = Curr_ptr->exe_time;
          new_order[node_index+1][j].breakdown_time = Curr_ptr-
>breakdown_time;
          new_order[node_index+1][j].proc_type = Curr_ptr->proc_type;
          new_order[node_index+1][j].start_time = Curr_ptr->start_time;
       }
    }

  node_index++;
  Head_ptr = &order[node_index][j];
  Tail_ptr = &order[node_index+1][j];
  Curr_ptr = Tail_ptr;

 }
```

```
   }
   node_input_file.close();
   order_output_file.close();
 }

// Calculate unused execution cycles
void
node_alloc::calc_unused_exe_cycles()
{

   node_type *Head_ptr, *Tail_ptr, *Curr_ptr;
   long unused_exe_cycles = 0;
   long total_unused_exe_cycles = 0;

   for (int j = 0; j < number_of_processors; j++ )
   {
    int node_index = 0;

    Head_ptr = &new_order[node_index][j];
    Tail_ptr = &new_order[node_index+1][j];
    Curr_ptr = Tail_ptr;

    unused_exe_cycles += Head_ptr->setup_time;

    while ( Tail_ptr->node_id != NULL )
    {
     int swap_index = node_index;

       if ( Head_ptr->exe_time < Tail_ptr->setup_time )
       {
           unused_exe_cycles += Tail_ptr->setup_time - Head_ptr->exe_time;
        }

         if ( Head_ptr->breakdown_time > Tail_ptr->exe_time )
         {
          int tail_index = swap_index + 2;
          Tail_ptr = &order[tail_index][j];
          unused_exe_cycles += Head_ptr->breakdown_time - Curr_ptr->exe_time +
Tail_ptr->setup_time + Curr_ptr->breakdown_time;
          node_index++;
         }
     node_index++;
     Head_ptr = &new_order[node_index][j];
     Tail_ptr = &new_order[node_index+1][j];
     Curr_ptr = Tail_ptr;
     }

         if ( Tail_ptr->node_id == NULL && Head_ptr->node_id != NULL )
         {                                                     // last node, add breakdown
              unused_exe_cycles += Head_ptr->breakdown_time;
         }

       total_unused_exe_cycles += unused_exe_cycles;
       unused_exe_cycles = 0;
   }

  cout << "Total Unused execution cycles are: " << total_unused_exe_cycles << " cycles "
<< newln;

}


// Function to implement the wrap-around
void
node_alloc::wrap_around()
{
        // Initialize values
  long Largest_cyl_time = 0, total_slice_times = 0;
  long idle_exe_cycles = 0, blocked_exe_cycles = 0;
  long idle_ctrl_cycles = 0, blocked_ctrl_cycles = 0;
  long blocked_proc_ctrl = 0, blocked_proc_exe = 0;
  long idle_proc_exe = 0, idle_proc_ctrl = 0;
  long total_idle_proc_exe = 0, total_idle_proc_ctrl = 0;
  long total_blocked_proc_exe = 0, total_blocked_proc_ctrl = 0;
  long exe_packing = 0, ctrl_packing = 0;

  node_type *Head_ptr, *Tail_ptr, *Next_ptr;

  ofstream cylTime_output_file;
  cylTime_output_file.open("CYLTIMES.OUT");
  if ( !cylTime_output_file )
  {
   cerr << "\nCannot Open file CYLTIMES.OUT\n";
   exit(0);
  }


  ofstream slice_output_file;
  slice_output_file.open("slice_time.out");
  if ( !slice_output_file )
  {
   cerr << "\nCannot Open file slice_time.out\n";
   exit(0);
  }


  // calculate the number of nodes on each processor

  for(int k = 0; k < number_of_processors; k++)
  {
    int index = 0;

    Head_ptr = &new_order[index][k];

   while ( Head_ptr->node_id != NULL )
   {
    index++;
    Head_ptr = &new_order[index][k];
    }
      new_order[0][k].nodes_per_processor = index;
   }


   for(int j = 0; j < number_of_processors; j++)
   {
     Head_ptr = &new_order[0][j]; //point to first node
     Head_ptr->start_time = 0;

     int numNodes = Head_ptr->nodes_per_processor;

     long cyl_time = Head_ptr->setup_time;
     blocked_proc_ctrl = 0;
     blocked_proc_exe = 0;

     int FLAG = 0, STUFFED = 0, PUSHED = 0;


     if(numNodes == 1)                    //Only one node on processor
     {
      cyl_time += (Head_ptr->exe_time + Head_ptr->breakdown_time);

           if(Head_ptr->breakdown_time + Head_ptr->setup_time < Head_ptr->exe_time)
           {
             Head_ptr->end_time = cyl_time - Head_ptr->breakdown_time;
             cyl_time -= Head_ptr->breakdown_time;
           }
           else
           {
              Head_ptr->end_time = cyl_time;
           }
         Largest_cyl_time = Head_ptr->end_time;
     }

    for(int i = 1; i < numNodes; i++)           // More than one node
    {
      Tail_ptr = &new_order[i][j]; // point to next node
      Tail_ptr->start_time = cyl_time;
      Next_ptr = &new_order[i+1][j];

//Several conditions are possible which modify the way the blocked cycles are calculated

         // Condition 1
          if(PUSHED && Next_ptr->node_id != 0)  //PUSHED =>A node's breakdown
          {                                     //was greater than another node's
            Head_ptr = &new_order[i][j];        //exe time
            Tail_ptr = &new_order[i+1][j];
            ++i;
            Next_ptr = &new_order[i+1][j];
            cyl_time += Head_ptr->setup_time;
            Tail_ptr->start_time = cyl_time;
            PUSHED = 0;
          }

        //Condition 2
        if (!FLAG && !PUSHED)    //FLAG=>
        {
          if (Head_ptr->exe_time > Tail_ptr->setup_time)
          {
                cyl_time += Head_ptr->exe_time;
                blocked_ctrl_cycles += Head_ptr->exe_time - Tail_ptr->setup_time;
                blocked_proc_ctrl += Head_ptr->exe_time - Tail_ptr->setup_time;
          }
          else
          {
                cyl_time += Tail_ptr->setup_time;
                blocked_exe_cycles += Tail_ptr->setup_time - Head_ptr->exe_time;
                blocked_proc_exe +=  Tail_ptr->setup_time - Head_ptr->exe_time;
           }
        }

        //Condition 3
          if(!STUFFED && !PUSHED) //STUFFED=> breakdown of a node and setup of
          {                               //next node occur during exe of another node
             Head_ptr->end_time = cyl_time + Head_ptr->breakdown_time;
          }

        //Condition 4
         if(Head_ptr->breakdown_time < Tail_ptr->exe_time && !PUSHED)
         {
            cyl_time += Tail_ptr->exe_time;
            FLAG = 0;
            Tail_ptr->end_time = cyl_time + Tail_ptr->breakdown_time;

        //Condition 5
          if((Head_ptr->breakdown_time + Next_ptr->setup_time) < Tail_ptr->exe_time
&& Next_ptr->node_id != 0)
          {
                FLAG = 1;
                STUFFED = 1;
                Next_ptr->start_time = cyl_time - Tail_ptr->exe_time + Tail_ptr-
>breakdown_time;
                blocked_ctrl_cycles += Tail_ptr->exe_time - Head_ptr->breakdown_time -
Next_ptr->setup_time;
                blocked_proc_ctrl += Tail_ptr->exe_time - Head_ptr->breakdown_time -
Next_ptr->setup_time;
          }
          else
          {
                if(Next_ptr->node_id != 0)
                {
                  STUFFED = 1;
                  FLAG = 1;
                  Next_ptr->start_time = cyl_time - Tail_ptr->exe_time + Tail_ptr-
>breakdown_time;
```

```
                    blocked_exe_cycles += Head_ptr->breakdown_time + Next_ptr-
>setup_time - Tail_ptr->exe_time;
                    blocked_proc_exe += Head_ptr->breakdown_time + Next_ptr-
>setup_time - Tail_ptr->exe_time;
                    cyl_time += blocked_proc_exe;
                    Tail_ptr->end_time += blocked_proc_exe;

                 }
             }
         }
         else
         {

           if(Next_ptr->node_id == 0 && PUSHED)
           {
             cyl_time += (Tail_ptr->setup_time + Tail_ptr->exe_time);
             Tail_ptr->end_time = cyl_time + Tail_ptr->breakdown_time;
             cyl_time = Tail_ptr->end_time;
           }
           else
           {
                 if(!PUSHED)
                 {
                  PUSHED = 1;
                  cyl_time += Head_ptr->breakdown_time;
                  Tail_ptr->end_time = cyl_time + Tail_ptr->breakdown_time;
                  cyl_time = Tail_ptr->end_time;
                  blocked_exe_cycles += Head_ptr->breakdown_time - Tail_ptr->exe_time;
                  blocked_proc_exe += Head_ptr->breakdown_time - Tail_ptr->exe_time;
                 }
           }
         }

        Head_ptr = &new_order[i][j];
    }

        if(numNodes != 1 && !PUSHED)
        {
          cyl_time += Tail_ptr->breakdown_time;
        }


      // Check for wrap-around condition
         Head_ptr = &new_order[0][j]; // Point to first node on processor
         Next_ptr = &new_order[1][j]; // Point to second node

      if((Tail_ptr->breakdown_time + Head_ptr->breakdown_time + Next_ptr-
>setup_time) <= (Head_ptr->exe_time + Next_ptr->exe_time) && Head_ptr-
>nodes_per_processor != 1 && !STUFFED && !PUSHED)
      {
          Tail_ptr->end_time = Head_ptr->setup_time + Next_ptr->setup_time + Tail_ptr-
>breakdown_time;
          cyl_time -= Tail_ptr->breakdown_time;
```

```
                 if (Next_ptr->setup_time + Tail_ptr->breakdown_time < Head_ptr-
>exe_time)
                 {
                   Head_ptr->end_time = Head_ptr->setup_time + Head_ptr->exe_time +
Head_ptr->breakdown_time;
                   blocked_ctrl_cycles -= Tail_ptr->breakdown_time;
                   blocked_proc_ctrl -= Tail_ptr->breakdown_time;
                 }
                 else
                Head_ptr->end_time = Tail_ptr->end_time + Head_ptr->breakdown_time;
        }
        else   // STUFFED or PUSHED
        {
         if(STUFFED)
         {
          node_type *Third_ptr = &new_order[2][j];

          if ((Tail_ptr->breakdown_time + Head_ptr->breakdown_time + Next_ptr-
>setup_time + Third_ptr->setup_time <= Head_ptr->exe_time + Next_ptr->exe_time) &&
Head_ptr->nodes_per_processor != 1)
          {
            Tail_ptr->end_time = Head_ptr->setup_time + Next_ptr->setup_time +
Tail_ptr->breakdown_time;
            cyl_time -= Tail_ptr->breakdown_time;
                 if (Next_ptr->setup_time + Tail_ptr->breakdown_time < Head_ptr-
>exe_time)
                 {
                   Head_ptr->end_time = Head_ptr->setup_time + Head_ptr->exe_time +
Head_ptr->breakdown_time;
                   blocked_ctrl_cycles -= Tail_ptr->breakdown_time;
                   blocked_proc_ctrl -= Tail_ptr->breakdown_time;
                 }
                 else
                 Head_ptr->end_time = Tail_ptr->end_time + Head_ptr->breakdown_time;
                 Third_ptr->start_time = Head_ptr->end_time;
          }
         }
         else
         {
          if(PUSHED)
          {
                 if(Next_ptr->setup_time + Tail_ptr->breakdown_time <= Head_ptr-
>exe_time)
                 {
                   Tail_ptr->end_time = Head_ptr->setup_time + Next_ptr->setup_time +
Tail_ptr->breakdown_time;
            cyl_time -= Tail_ptr->breakdown_time;
                 }
          }
         }
        }

   total_slice_times += cyl_time;      // add all cylinder times
   slice_output_file << cyl_time << endl;
```

```
      total_blocked_proc_ctrl += blocked_proc_ctrl;
      total_blocked_proc_exe += blocked_proc_exe;

      cylTime_output_file << j+1 << setw(8) << cyl_time << endl;

      Head_ptr = &new_order[0][j];

      exe_packing = 0, ctrl_packing = 0;

      for ( int p = 0; p < numNodes; p++ )        //Calculate exe and control unit packing
      {
           Head_ptr = &new_order[p][j];
           exe_packing += Head_ptr->exe_time;
           ctrl_packing += Head_ptr->setup_time + Head_ptr->breakdown_time;
      }

          //Calculate idle cycle times
      idle_proc_exe = cyl_time - exe_packing - blocked_proc_exe;
      idle_proc_ctrl = cyl_time - ctrl_packing - blocked_proc_ctrl;
      total_idle_proc_exe += idle_proc_exe;
      total_idle_proc_ctrl += idle_proc_ctrl;

      generate_processor_stats(j+1, cyl_time, exe_packing, ctrl_packing, idle_proc_exe,
idle_proc_ctrl, blocked_proc_exe, blocked_proc_ctrl);

    slice_output_file.close();

    }

// Find the largest end time for all processors for "flat" cylinder

  for(int m = 0; m < number_of_processors; m++)
   {
     int index = 0;

     Head_ptr = &new_order[index][m];

    while ( Head_ptr->node_id != NULL )
    {
     index++;

         if(Largest_cyl_time < Head_ptr->end_time)
         {
           Largest_cyl_time = Head_ptr->end_time;
         }
         cylTime_output_file << endl << endl << Largest_cyl_time << endl;

     Head_ptr = &new_order[index][m];
    }

   }
         // Find idle time for "flat" cylinder
         idle_ctrl_cycles = (Largest_cyl_time*number_of_processors) - blocked_ctrl_cycles;
```

```
          idle_exe_cycles = (Largest_cyl_time*number_of_processors) -
blocked_exe_cycles;


       make_cylinder_file(Largest_cyl_time);
          generate_statistics(Largest_cyl_time,idle_ctrl_cycles, blocked_ctrl_cycles,
idle_exe_cycles, blocked_exe_cycles, total_idle_proc_exe, total_idle_proc_ctrl,
total_slice_times);

          cylTime_output_file.close();
}


//Function to create the cylinder output file
void
node_alloc::make_cylinder_file(long Largest_cyl_time)
{

  node_type *Head_ptr;

  ofstream cylinder_output_file;
  cylinder_output_file.open("cylinder.out");
  if ( !cylinder_output_file )
  {
   cerr << "\nCannot Open file cylinder.out\n";
   exit(0);
  }

  for ( int j = 0; j < number_of_processors; j++ ) //Print out node order
  {

    Head_ptr = &new_order[0][j];
    int processorNodes = Head_ptr->nodes_per_processor;
    cylinder_output_file << newln << processorNodes << endl << endl;

    for ( int i = 0; i < processorNodes; i++ )
    {
         Head_ptr = &new_order[i][j];
         cylinder_output_file << setw(7) << Head_ptr->node_id;
         cylinder_output_file << setw(12) << Head_ptr->start_time;
         cylinder_output_file << setw(12) << Head_ptr->end_time;
         cylinder_output_file << endl;
     }
  }

  cylinder_output_file << newln << newln << Largest_cyl_time << endl;
  cylinder_output_file.close();
}


// Function to print individual processor statistics
void
```

```
node_alloc::generate_processor_stats(int procNum, long cyl_time, long exe_packing, long
ctrl_packing, long idle_proc_exe, long idle_proc_ctrl, long blocked_proc_exe, long
blocked_proc_ctrl)
{

  ofstream processor_stats_file;
  processor_stats_file.open("proc_stats.out", ios::app);
  if ( !processor_stats_file )
  {
   cerr << "\nCannot Open file proc_stats.out\n";
   exit(0);
  }

  processor_stats_file << "PROCESSOR UTILIZATION\n\n";
  processor_stats_file << "NUMBER OF PROCESSORS : ";
  processor_stats_file << setw(4) << number_of_processors << endl << endl;
  processor_stats_file << "CYCLES PER WORD      : ";
  processor_stats_file << setw(4) << comm << endl << endl;


  double ctrl_util_rate = (double)ctrl_packing/cyl_time*100.0;
  double ctrl_idle_rate = (double)idle_proc_ctrl/cyl_time*100.0;
  double ctrl_blocked_rate = (double)blocked_proc_ctrl/cyl_time*100.0;

  double exe_util_rate = (double)exe_packing/cyl_time*100.0;
  double exe_idle_rate = (double)idle_proc_exe/cyl_time*100.0;
  double exe_blocked_rate = (double)blocked_proc_exe/cyl_time*100.0;

  processor_stats_file << "PROCESSOR NUMBER     :";
  processor_stats_file << setw(4) << procNum << endl;

  processor_stats_file << "\nCONTROL UNIT UTILIZATION\n\n";

  processor_stats_file.setf(ios::fixed);
  processor_stats_file.setf(ios::showpoint);
  processor_stats_file << "BEST CYLINDER PACKING ( CONTROL TIME )    : ";
  processor_stats_file << setw(12) << ctrl_packing << endl << endl;
  processor_stats_file << "END TIME OF ACTIVITIES                    : ";
  processor_stats_file << setw(12) << cyl_time << endl << endl;
  processor_stats_file << "Control Unit Utilization Rate      : ";
  processor_stats_file << setw(6) << setprecision(1);
  processor_stats_file << ctrl_util_rate << "%\n";
  processor_stats_file << "Control Unit Idle Rate             : ";
  processor_stats_file << setw(6) << setprecision(1);
  processor_stats_file << ctrl_idle_rate << "%\n";
  processor_stats_file << "Control Unit Blockage Rate         : ";
  processor_stats_file << setw(6) << setprecision(1);
  processor_stats_file << ctrl_blocked_rate << "%\n\n\n\n\n";

  processor_stats_file << "EXECUTION UNIT UTILIZATION\n\n";

  processor_stats_file << "BEST CYLINDER PACKING ( EXECUTION TIME ) : ";
  processor_stats_file << setw(12) << exe_packing << endl << endl;
  processor_stats_file << "END TIME OF ACTIVITIES                   : ";
```

```
    processor_stats_file << setw(12) << cyl_time << endl << endl;
    processor_stats_file << "Execution Unit Utilization Rate    : ";
    processor_stats_file << setw(6) << setprecision(1);
    processor_stats_file << exe_util_rate << "%\n";
    processor_stats_file << "Execution Unit Idle Rate           : ";
    processor_stats_file << setw(6) << setprecision(1);
    processor_stats_file << exe_idle_rate << "%\n";
    processor_stats_file << "Execution Unit Blockage Rate       : ";
    processor_stats_file << setw(6) << setprecision(1);
    processor_stats_file << exe_blocked_rate << "%\n\n\n";
    processor_stats_file << endl;

  }


// Function to print cylinder statistics
void
node_alloc::generate_statistics(long Largest_cyl_time, long idle_ctrl_cycles, long
blocked_ctrl_cycles, long idle_exe_cycles, long blocked_exe_cycles, long
total_idle_proc_exe, long total_idle_proc_ctrl, long total_slice_times)
{
  long exe_cyl_packing = 0;
  long ctrl_cyl_packing = 0;

  node_type *Head_ptr;

  ofstream statistics_output_file;
  statistics_output_file.open("cyl_stats.out");
  if ( !statistics_output_file )
  {
   cerr << "\nCannot Open file cyl_stats.out\n";
   exit(0);
  }

  statistics_output_file << "PROCESSOR UTILIZATION\n\n";
  statistics_output_file << "NUMBER OF PROCESSORS : ";
  statistics_output_file << setw(4) << number_of_processors << endl << endl;
  statistics_output_file << "CYCLES PER WORD      : ";
  statistics_output_file << setw(4) << comm << endl << endl;


  for(int j = 0; j < number_of_processors; j++)
  {
    Head_ptr = &new_order[0][j];
    int processorNodes = Head_ptr->nodes_per_processor;

    for ( int i = 0; i < processorNodes; i++ )  //Calculate exe and ctrl unit packing per
    {                                           //processor
        Head_ptr = &new_order[i][j];
        exe_cyl_packing += Head_ptr->exe_time;
        ctrl_cyl_packing += Head_ptr->setup_time + Head_ptr->breakdown_time;
    }
  }
      //Calculate values for "jagged" cylinder
```

```
 long avg_slice_time = total_slice_times/number_of_processors;
 long best_exe_packing = exe_cyl_packing/number_of_processors;
 long best_ctrl_packing = ctrl_cyl_packing/number_of_processors;
 long avg_ctrl_idle = (idle_ctrl_cycles/number_of_processors) - best_ctrl_packing;
 long avg_ctrl_blocked = blocked_ctrl_cycles/number_of_processors;
 long avg_exe_idle = (idle_exe_cycles/number_of_processors) - best_exe_packing;
 long avg_exe_blocked = blocked_exe_cycles/number_of_processors;
 long avg_ctrl_idle_jag = (total_idle_proc_ctrl/number_of_processors) -
best_ctrl_packing;
 if(avg_ctrl_idle_jag < 0)
   avg_ctrl_idle_jag = 0;
 long avg_exe_idle_jag = (total_idle_proc_exe/number_of_processors) -
best_exe_packing;
 if(avg_exe_idle_jag < 0)
   avg_exe_idle_jag = 0;

       //Calculate "flat" and "jagged" cylinder statistics
 double exe_util_rate = (double)best_exe_packing/Largest_cyl_time*100.0;
 double ctrl_util_rate = (double)best_ctrl_packing/Largest_cyl_time*100.0;
 double ctrl_idle_rate = (double)avg_ctrl_idle/Largest_cyl_time*100.0;
 double ctrl_blocked_rate = (double)avg_ctrl_blocked/Largest_cyl_time*100.0;
 double exe_idle_rate = (double)avg_exe_idle/Largest_cyl_time*100.0;
 double exe_blocked_rate = (double)avg_exe_blocked/Largest_cyl_time*100.0;
 double ctrl_idle_rate_jag = (double)avg_ctrl_idle_jag/avg_slice_time*100.0;
 double ctrl_util_rate_jag = (double)best_ctrl_packing/avg_slice_time*100.0;
 double ctrl_blocked_rate_jag = (double)avg_ctrl_blocked/avg_slice_time*100.0;
 double exe_util_rate_jag = (double)best_exe_packing/avg_slice_time*100.0;
 double exe_idle_rate_jag = (double)avg_exe_idle_jag/avg_slice_time*100.0;
 double exe_blocked_rate_jag = (double)avg_exe_blocked/avg_slice_time*100.0;


 statistics_output_file << "\n\n\nCONTROL UNIT UTILIZATION\n\n";

 statistics_output_file.setf(ios::fixed);
 statistics_output_file.setf(ios::showpoint);
 statistics_output_file << "BEST CYLINDER PACKING ( CONTROL TIME )   : ";
 statistics_output_file << setw(12) << best_ctrl_packing << endl << endl;
 statistics_output_file << "END TIME OF ACTIVITIES ('FLAT' CYLINDER)  : ";
 statistics_output_file << setw(12) << Largest_cyl_time << endl << endl;
 statistics_output_file << "Control Unit Utilization Rate      : ";
 statistics_output_file << setw(6) << setprecision(1);
 statistics_output_file << ctrl_util_rate << "%\n";
 statistics_output_file << "Control Unit Idle Rate             : ";
 statistics_output_file << setw(6) << setprecision(1);
 statistics_output_file << ctrl_idle_rate << "%\n";
 statistics_output_file << "Control Unit Blockage Rate         : ";
 statistics_output_file << setw(6) << setprecision(1);
 statistics_output_file << ctrl_blocked_rate << "%\n\n\n";
 statistics_output_file << "END TIME OF ACTIVITIES ('JAGGED' CYLINDER): ";
 statistics_output_file << setw(12) << avg_slice_time << endl << endl;
 statistics_output_file << "Control Unit Utilization Rate      : ";
 statistics_output_file << setw(6) << setprecision(1);
 statistics_output_file << ctrl_util_rate_jag << "%\n";
 statistics_output_file << "Control Unit Idle Rate             : ";
```

```
    statistics_output_file << setw(6) << setprecision(1);
    statistics_output_file << ctrl_idle_rate_jag << "%\n";
    statistics_output_file << "Control Unit Blockage Rate        : ";
    statistics_output_file << setw(6) << setprecision(1);
    statistics_output_file << ctrl_blocked_rate_jag << "%\n\n\n\n\n\n";


    statistics_output_file << "EXECUTION UNIT UTILIZATION\n\n";

    statistics_output_file << "BEST CYLINDER PACKING ( EXECUTION TIME ) : ";
    statistics_output_file << setw(12) << best_exe_packing << endl << endl;
    statistics_output_file << "END TIME OF ACTIVITIES ('FLAT' CYLINDER) : ";
    statistics_output_file << setw(12) << Largest_cyl_time << endl << endl;
    statistics_output_file << "Execution Unit Utilization Rate   : ";
    statistics_output_file << setw(6) << setprecision(1);
    statistics_output_file << exe_util_rate << "%\n";
    statistics_output_file << "Execution Unit Idle Rate          : ";
    statistics_output_file << setw(6) << setprecision(1);
    statistics_output_file << exe_idle_rate << "%\n";
    statistics_output_file << "Execution Unit Blockage Rate      : ";
    statistics_output_file << setw(6) << setprecision(1);
    statistics_output_file << exe_blocked_rate << "%\n\n\n";
    statistics_output_file << "END TIME OF ACTIVITIES ('JAGGED' CYLINDER): ";
    statistics_output_file << setw(12) << avg_slice_time << endl << endl;
    statistics_output_file << "Execution Unit Utilization Rate   : ";
    statistics_output_file << setw(6) << setprecision(1);
    statistics_output_file << exe_util_rate_jag << "%\n";
    statistics_output_file << "Execution Unit Idle Rate          : ";
    statistics_output_file << setw(6) << setprecision(1);
    statistics_output_file << exe_idle_rate_jag << "%\n";
    statistics_output_file << "Execution Unit Blockage Rate      : ";
    statistics_output_file << setw(6) << setprecision(1);
    statistics_output_file << exe_blocked_rate_jag << "%\n\n";
    statistics_output_f[illegible] << flush;


}


// Function to reorder the ORDER.IN file sequentially
// This function uses a simple bubble algorithm to reorder the ORDER.IN file
void
node_alloc::sequence_nodes()
{
  int i, SWAPPED;
  order_in_type  TEMP_NODE;

  ifstream order_input_file;
  order_input_file.open("ORDER.IN");

  ifstream node_input_file;
  node_input_file.open("NODES.IN");
  node_input_file >> number_of_nodes;
  node_input_file.close();

  ofstream order_output_file;
  order_output_file.open("TEMP_ORDER.IN");

  for ( int count = 0; count < number_of_nodes; count++ )
  {
    order_input_file >> ORDER[count].node_id;
    order_input_file >> ORDER[count].start_time;
  }

  // Order Nodes in order of increasing start time

   SWAPPED = 1;
   while ( SWAPPED )
   {
       SWAPPED = 0;
       for ( i = 0; i < number_of_nodes; i++ )
       {
         if ( ORDER[i].start_time > ORDER[i+1].start_time ) // Reorder nodes
         {
              TEMP_NODE = ORDER[i];
              ORDER[i] = ORDER[i+1];
              ORDER[i+1] = TEMP_NODE;
              SWAPPED = 1;
       }
        }
   }
       //Put reordered nodes into output file
    for ( i = 0; i <= number_of_nodes; i++ )
    {
      if ( ORDER[i].node_id != NULL )
      {
           order_output_file << ORDER[i].node_id << setw(8) << "0" <<newln;
         }
    }

  order_input_file.close();
  order_output_file.close();
  system("mv ORDER.IN ORDER_IN.ORG");
  system("mv TEMP_ORDER.IN ORDER.IN");
  system("mv NODES.IN NODES_SNB.IN");
  system("mv NODES_IN.ORG NODES.IN");

}

// end of program
```

# APPENDIX B: PROGRAM USER'S MANUAL

## I. NODE SCHEDULING PROGRAM

This section describes a Large Grain Data Flow node-to-processor scheduling program (referred to as SCHEDULE) which provides a detailed node-to-processor scheduling of a data flow graph using the model described in [Ref. 4]. The program uses a two dimensional array to represent the revolving cylinder to generate the order the nodes should enter the system based on input data files and data provided by the user. The program also determines if the breakdown time of the last node on a processor can be 'wrapped-around' to provide an accurate modeling of the revolving cylinder. This mapping is only concerned with the arithmetic processors and the program nodes. Therefore, input and output nodes and the input/output processors described in [Ref. 4] are not included in this scheduling program or associated data files. This program must be run prior to executing the mapping program discussed in Section II. This program begins execution with the command 'schedule'.

### A. USER INTERFACE

The following inputs and options are available to the user:

#### 1. SCHEDULER LATENCY TIME

A number which abstractly represents the time it takes the scheduler to change the state of its local memory when amounts on a queue are modified due to node input or output.

### 2. COMMUNICATION TIME FOR ONE WORD

This is the time to transmit one word of data between a memory module and a processor.

## B. INPUT FILES

### 1. Input File: NODES.IN

This file contains the initial node information required for mapping. The number of nodes parameter is an individual element. The remaining parameters exist for each node in the graph.

#### *a. Number of Nodes*

This is the total number of nodes in the data flow graph.

#### *b. Node ID*

This is the node identifier number.

#### *c. Instruction Size*

This is the node instruction size parameter in words.

#### *d. Setup Time*

This is the node setup time in cycles.

#### *e. Execution Time*

This is the node execution time parameter in cycles.

#### *f. Breakdown Time*

This is the node breakdown time parameter in cycles.

#### *g. Processor Type*

This is the processor number that the node will be assigned to.

### 2. Input File: QUEUES.IN

This file contains the initial queue information required for scheduling. The number of queues parameter is an individual element. The remaining parameters exist for each queue in the graph.

#### *a. Number of Queues*

This is the total number of queues in the data flow graph.

#### *b. Queue ID*

This is the queue identifier number.

#### *c. Source Node*

This is the node ID for the node at the tail of the queue.

#### *d. Sink Node*

This is the node ID for the node at the head of the queue.

#### *e. Write Amount*

This is the queue write amount parameter in words.

#### *f. Read Amount*

This is the queue read amount parameter in words.

### 2. Input File: PROCS.IN

This file is fully described in Section II. The only data taken from this file is the number of processors parameter.

## C. OUTPUT FILES

Many output files are created for input to the mapping program.

### 1. Output File: ORDER.IN

This file is the mapping order of the nodes. The mapping occurs in the order the nodes are listed.

#### *a. Node ID*

This is the node identifier of the next node to enter the system.

#### *b. Time into System*

This is the time when the node will be available to be mapped. Normally, all nodes will have a time of '0' which means all nodes are available to be mapped simultaneously from the start time.

### 2. Output File: NODES_SNB.OUT

This file is similar in format to the NODES.IN file but also contains the calculated values of setup and breakdown for the nodes in the system based on the user input. This file is not used by the mapper program; it is for user information only.

### 3. Output File: cylinder.out

This file is a representation of the mapping of the cylinder. It is in the same format as the file 'cylinder.dat' which is described fully as an input to the synchronization arc generator (SAG) program, however, this file takes into account the possibility of 'wrap-around' of the breakdown of the last node on a processor. The name of this file must be changed to 'cylinder.dat' before using it for input to the SAG program.

### 4. Output File: cyl_stats.out

In this file are several percentages to express the efficiency of the mapping. Two sets of statistics are given. In the first, the largest completion time over all processors is computed and all processors are assumed to run to this time ("flat cylinder"). The statistics are then computed over the total processor-time required by the mapping. This is the largest completion time over all processors multiplied by the number of processors. In the second set ("jagged cylinder"), each processor completion time is calculated individually and the statistics computed for each processor, the average is then taken of the individual processor statistics .

#### *a. Control Unit and Execution Unit Utilization Rate*

This refers to the total percentage of processor-time that the specified unit (control or execution) is performing useful work, either input or output for the control unit or execution for the execution unit.

#### *b. Control Unit and Execution Unit Blockage Rate*

This refers to the total percentage of processor-time that the specified unit (control or execution) is blocked, i.e., the unit has completed the specific task, but the node cannot switch to the other unit as the other unit is currently busy.

#### c. *Control Unit and Execution Unit Idle Rate*

This refers to the total percentage of processor-time that the specified unit (control or execution) has no node assigned.

### 5. Output File: proc_stats.out

The processor statistics are outlined in this file. The statistic listings are essentially the same as for 'cyl_stats.out', except that the statistics are computed over one processor vice an average over all processors. Each processor is also treated as a 'jagged' slice, that is, no attempt is made to find the greatest completion time of all processor slices; the statistics are calculated based on the final completion time for each individual processor.

# II. LGDF MAPPING PROGRAM

This section describes a Large Grain Data Flow mapping program (referred to as MAP) which provides a detailed multiprocessor mapping of a data flow graph using the model described in [Ref. 4]. The program is time driven. As events are scheduled to occur, the event with the lowest time stamp will set the next time flag. When this flagged time occurs, all nodes are checked for the next event to occur. A set of lists track which nodes are in the various states of processing. This mapping is only concerned with the arithmetic processors and the program nodes. Therefore, input and output nodes and the input/output processors described in [Ref. 4] are not included in this mapping program or associated data files. This program must be run prior to executing the synchronization arc generator program or the simulator program discussed in Sections III and IV, respectively. This program begins execution with the command 'map'.

## A. USER INTERFACE

The following inputs and options are available to the user:

### 1. SCHEDULER LATENCY TIME

### COMMUNICATION TIME FOR ONE WORD

These inputs were fully discussed in Section I.

### 2. INTERACTIVE INTERFACE

The user can select whether or not to use the interactive interface. The interface will allow the user to see the current state of the system at any time. Also, the user can adjust the operation of the system by manipulating nodes which are waiting to begin processing.

## B. INPUT FILES

### 1. Input File: NODES.IN

### Input File: QUEUES.IN

These files were also previously described in Section I.

### 2. Input File: CHAINS.IN

This file contains the initial chain information required for mapping. The number of chains parameter is an individual element. The remaining parameters exist for each chain in the graph. Note that this file is required to exist, or execution will fail. If there are no chains, then simply have '0' as the only entry in the file.

#### *a. Number of Chains*

This is the total number of chains in the system.

#### *b. Chain ID*

This is the chain identifier number.

#### *c. Chained Nodes*

The node IDs for the nodes included in the chain are listed in the order of chaining. A '0' is used to identify the end of the node list for the chain.

### 3. Input File: PROCS.IN

The following information describes the hardware configuration.

#### *a. Number of Arithmetic Processors*

This is the total number of arithmetic processors in the system.

#### *b. Processor Type*

The processor type is listed for the number of processors in the system. For example, if there are three processors, the numbers 1, 2, and 3 will be listed in a single column.

### 4. Input File: ORDER.IN

This file is the mapping order of the nodes. The mapping occurs in the order the nodes are listed. This file can be created manually by the user or can be generated using the scheduler program. This file is fully described as an output to the scheduler program in Section I.

## C. OUTPUT FILES

Many output files are created for complete information on the mapping.

### 1. Output Files: CON_EXE.OUT,CON_UNIT.OUT,EXE_UNIT.OUT

These three files provide an exact mapping of the nodes on the processors. The events occurring at a specific time and the nodes involved are depicted. A key to the markings is listed in each file. File 'CON_EXE.OUT' provides a complete mapping file, file 'EXE_UNIT.OUT' is a mapping of the execution units only and the file 'CON_UNIT.OUT' is a mapping of the control units only. These output file listings do not take into account 'wrap-around' of the last node's breakdown time.

In each file are several percentages to express the efficiency of the mapping. An important note about the statistics is that they are computed over the total processor-time required by the mapping. This is the time to complete the mapping multiplied by the number of processors. The percentages are therefore essentially an average of the individual processor rates.

#### *a. Processor Utilization Rate*

This refers to the total percentage of processor-time that a processor is performing some activity in either the control unit or the execution unit.

#### *b. Processor Idle Rate*

This refers to the total percentage of processor-time that a processor is not performing any activity.

#### *c. Control Unit and Execution Unit Utilization Rate*

This refers to the total percentage of processor-time that the specified unit (control or execution) is performing useful work, either input or output for the control unit or execution for the execution unit.

#### *d. Control Unit and Execution Unit Blockage Rate*

This refers to the total percentage of processor-time that the specified unit (control or execution) is blocked, i.e., the unit has completed the specific task, but the node cannot switch to the other unit as the other unit is currently busy.

#### *e. Control Unit and Execution Unit Idle Rate*

This refers to the total percentage of processor-time that the specified unit (control or execution) has no node assigned.

### 2. Output File: SUMMARY.OUT

This file summarizes the number of processors in particular states at any given time. The event times in the three previous mapping files will match with this file. The processor utilization percentages are displayed.

### 3. Output Files: NODES.OUT, PROCS.OUT, CHAINS.OUT

These three files provide extremely detailed data on specific nodes, processors, and chains. The lines are well described within the output listings. Most of the items can be cross-referenced to other files.

### 4. Output File: cylinder.dat

This file is a representation of the mapping of the cylinder. It is described fully as an input to the synchronization arc generator (SAG) program. The inclusion of this file is to provide the data necessary to run SAG based on the mapping generated by this program without any adjustments.

## D. SELECTION OF THE USER INTERFACE OPTION

The selection of the user interface option will allow the user to observe and interactively change the mapping as it progresses. However, once the mapping has progressed past an event, it is not possible to go back and make a change. The interactive interface is very descriptive. The user can view many aspects of the system and make many changes during any pause. Selecting the 'CONTINUE WITH NEXT EVENT' will allow the mapping to continue. To discontinue the use of the interactive interface, select the 'CHANGE INTERRUPT STRATEGY' followed by 'CONTINUE TO CONCLUSION' followed by 'CONTINUE WITH EVENT' options. This will allow the mapping to complete.

# III. LGDF SYNCHRONIZATION ARC GENERATOR

This section describes a Large Grain Data Flow model synchronization arc generator program (referred to as SAG). This program acts as a preprocessor to the simulator program (SIM). Its purpose is to modify the input files to SIM to be able to analyze the revolving cylinder (RC) method as described in [Ref. 4]. SAG makes extensive use of linked lists. SAG is started with the command 'generate'.

## A. USER INTERFACE

The user has a choice of one of two arc generation techniques in SAG. Both techniques are described fully in [Ref. 4].

### 1. Start After Finish (SAF)

This selection will determine the synchronization arcs based on the start after finish technique.

### 2. Start After Start (SAS)

This selection will determine the synchronization arcs based on the start after start technique.

## B. INPUT FILES

### 1. Input File: nodes.dat

This file is a tabular listing which completely describes the nodes of a data flow graph. The number of nodes parameter is an individual element. The remaining parameters exist for each node in the graph.

#### *a. Number of Nodes*

This is the total number of nodes in the data flow graph. This initializes the counters necessary to read in the node data.

*b. Node ID*

This is the node identifier number, which must be unique for each node in the system. Do not use '0' as a node ID.

*c. Node Type*

This identifies the type of node. This type defines how the node will be handled in the programs.

(1) node type = 0: normal node

(2) node type = 1: input node

(3) node type = 2: output node

*d. Instruction Size*

This is the node instruction size parameter in words.

*e. Execution Time*

This is the node execution time parameter in cycles.

*f. Setup Time*

This is the node setup time parameter in cycles.

*g. Breakdown Time*

This is the node breakdown time parameter in cycles.

*h. Required Processor Type*

This is the type of processor required by the node. A listing of '100' identifies an input/output processor.

*i. Alternate Processor Type*

This is the alternate processor type to be used if the required processor type is unavailable. In most cases, the alternate is the same as the required processor type.

*j. Memory Module Assignment*

This is the memory module assignment for the node if the user defined memory assignment option is chosen.

#### *k. Node Priority*

This is the assignment priority associated with the node if the user defined priority option is chosen. A lower number represents a higher priority.

### 2. Input File: queues.dat

This file is a tabular listing which completely describes the queues of a data flow graph. The number of queues parameter is an individual element. The remaining parameters exist for each queue in the graph.

#### *a. Number of Queues*

This is the total number of queues in the system. This initializes the counters necessary to read in the queue data.

#### *b. Queue ID*

This is the queue identifier number, which must be unique for each queue in the system. Do not use '0' as a queue ID.

#### *c. Queue Type*

This identifies the type of queue. The type defines how the queue will be handled in the programs.

(1) queue type = 0: data queue

(2) queue type = 1: input queue

(3) queue type = 2: output queue

(4) queue type = 3: synchronization arc

#### *d. Source Node*

This is the node ID for the node at the tail of the queue.

#### *e. Sink Node*

This is the node ID for the node at the head of the queue.

#### *f. Write Amount*

This is the queue write amount parameter in words.

*g. Read Amount*

This is the queue read amount parameter in words.

*h. Produce Amount*

This is the queue produce amount parameter in words.

*i. Consume Amount*

This is the queue consume amount parameter in words.

*j. Threshold Amount*

This is the queue threshold amount parameter in words.

*k. Initial Length*

This is the queue initial length parameter in words.

*l. Capacity*

This is the queue capacity parameter in words.

*m. Memory Module Assignment*

This is the memory module assignment for the queue if the user defined memory assignment option is chosen.

### 3. Input File: machine.cfg

This file defines the system hardware configuration.

*a. Number of Memory Modules*

This is the number of memory modules to be modeled in the simulator.

*b. Number of Input / Output Processors*

This is the number of input / output (I/O) processors to be modeled in the simulator. Normally there is only one I/O processor.

*c. Number of Arithmetic Processors*

This is the number of arithmetic processors in the system.

*d. Processor Types*

This is a list of the types of processors defined, with the number of elements in the list equal to the number of processors, excluding I/O processors which are automatically defined as '100'. If synchronization arcs without nodes bound to processors are desired, the user should enter a '0' for each element. If however, the user desires synchronization arcs generated with nodes bound to processors, each element should correspond to a processor type. For example, if there are three processors, the numbers 1, 2, and 3 should be listed in a column.

### 4. Input File: cylinder.dat

This file is a representation of the mapping of nodes on the processors. If an analysis of a cylinder with no wrap-around is desired, this file will be generated by the external mapping program (MAP). If an analysis of a cylinder with wrap-around is desired, this file is generated by the scheduler program, after the filename is modified from 'cylinder.out'.

#### *a. Number of Nodes on a Processor*

For each arithmetic processor in the system, the number of nodes which used that processor are given. Following the node total, the following data is provided for each node on the given processor:

(1) Node ID

(2) The node start time on the processor

(3) The node finish time on the processor

#### *b. Cylinder Size*

Following the listing of the nodes, the time to complete the cylinder slice is given. This is equal to the longest processor busy time of all the processors in the system.

## C. OUTPUT FILES

Many output files are created for complete information on the mapping.

### 1. Output File: queues.dat

This file has the same format as described previously for 'queues.dat'. However, synchronization arcs have been appended to the end of the file as determined by this program. This adjusted 'queues.dat' file may now be used by the simulator to analyze the revolving cylinder (RC) scheduling techniques.

**2. Output File: oqueues.dat**

This file is a copy of the original 'queues.dat'. Since this program modifies the file 'queues.dat', this file will allow for easy recovery back to the original graph description, prior to the addition of synchronization arcs.

**3. Output File: indexcyl.out**

This file provides the same information as the 'cylinder.dat' file. In addition, the appropriate index for the node is provided as described in [Ref. 4].

**4. Output File: tokens.out**

This file lists important information about the synchronization arcs, including the source node, sink node, initial length (number of tokens), threshold amount, consume amount, and produce amount.

**5. Temporary File: rqueues.tmp**

This file is a temporary file created during execution which will provide no useful information to the user.

# IV. LGDF SIMULATOR

This section describes a simulator (referred to as SIM) for a Large Grain Data Flow model described in [Ref. 4]. SIM is an event-driven program that makes extensive use of linked lists. SIM is started with the command 'simulate'.

## A. USER INTERFACE

There are many inputs and options available to the user. They are presented below exactly as they appear in the program.

### 1. COMMENT LINE

This is a comment which will be displayed at the head of the data set in the statistics file to enable the user to easily distinguish the file output. Results from successive executions of SIM can be dumped to a single file without losing track of the data sets.

### 2. THE INSTANCE NUMBER TO START GATHERING RESULTS

This is the input instance of the graph to start gathering throughput and utilization results from the simulation.

### 3. THE INSTANCE NUMBER TO TERMINATE THE SIMULATION

This is the output instance, which when completed, will terminate the simulation.

### 4. SCHEDULER LATENCY TIME (cycles)

This is scheduler latency for any queue variations in the scheduler internal memory. This could be the time taken by the scheduler to manipulate its internal data structures.

### 5. COMMUNICATION TIME FOR ONE WORD (cycles)

This is the time to transmit one word of data between a memory unit and a processor across the data transfer network.

### 6. DATA RATE OPTION

Two options are available:

#### *a. User Defined*

The user will be prompted for further input of the time interval which will pass after the input data for one graph iteration are entered into the system until the input data for the next graph iteration are entered into the system. The prompt seen by the user is: ENTER THE DATA PERIOD BEFORE THE NEXT GRAPH ITERATION (cycles).

#### *b. Maximum Throughput*

The simulator will generate data for consecutive graph iterations to insure that the input queue is constantly filled. This will drive the machine at its maximum throughput. This effectively permits the user to determine the upper bound in the input data rate for the given configuration.

### 7. MEMORY MAPPING OPTIONS

Two options are available:

#### *a. User Defined Mapping*

This option will map nodes and queues to memory modules as defined in the nodes.dat file.

#### *b. Arbitrary Mapping*

The simulator will arbitrarily assign nodes and queues to memory modules.

### 8. NODES ON READY LIST OPTION

Two options are available:

#### *a. Only One Node Instance can be on Ready List*

Only one instance of a node can be maintained on the ready list at any given time.

#### *b. Multiple Node Instances can be on Ready List*

Multiple instances of a node can be maintained on the ready list at any given time. However, only one instance of the node can be processing.

### 9. NODES EXECUTION PRIORITY OPTIONS

Several options are available to place nodes in the ready list.:

#### *a. No Priority*

Nodes are executed on a First-Come-First-Served (FCFS) basis, i.e., according to the order in which they are ready.

#### *b. User Defined*

The node priorities are as defined in the file 'nodes.dat' This allows the user to designate critical nodes to be assigned to a processor immediately when data is available.

#### *c. Shortest Execution Time First*

A ready node with a shorter execution time will be assigned before a ready node with a longer execution time.

#### *d. Longest Execution Time First*

A ready node with a longer execution time will be assigned before a ready node with a shorter execution time.

## B. INPUT FILES

Three input files are required by the simulator.

### 1. Input File: nodes.dat

The contents of this input file are described fully as an input to the synchronization arc generator program in Section 4.

### 2. Input File: queues.dat

The contents of this input file are described fully as an input to the synchronization arc generator program in Section 4.

### 3. Input File: machine.cfg

The contents of this input file are described fully as an input to the synchronization arc generator program in Section 4.

## C. OUTPUT FILES

Three output files are generated by this program.

### 1. Output Files: starts.out

This is a listing of the graph instance and the start times of those instances being measured.

### 2. Output File: endtimes.out

This is a listing of the graph instance and end times of those instances being measured.

### 3. Output Files: stats.out

This file summarizes the data from a given simulation. The same information is displayed to the standard output upon program completion. Note that this file is an appended file, so additional simulation results are added to the end of the file which enables easier comparison of multiple tests. The following data is provided:

#### *a. COMMENT*

The comment line input by the user.

#### *b. DATA RATE OPTION*

The number corresponding to the choice made at the start of the program.

#### *c. MEMORY MAPPING OPTION*

The number corresponding to the choice made at the start of the program.

#### *d. NODES ON READY LIST OPTION*

The number corresponding to the choice made at the start of the program.

#### *e. NODES EXECUTION PRIORITY OPTION*

The number corresponding to the choice made at the start of the program.

#### *f. START INSTANCE*

The data flow graph instance where measurements were started.

#### *g. END INSTANCE*

The data flow graph instance which terminated the simulation.

### *h. SCHEDULER LATENCY TIME (cycles)*

The scheduler latency for any queue adjustment in the scheduler internal memory.

### *i. COMMUNICATION TIME FOR ONE WORD (cycles)*

The communication time to transfer one word of data between a memory module and a processor.

### *j. ITERATION DATA PERIOD (cycles)*

The time differential for the input of consecutive data flow graph iterations, or the statement 'MAX THROUGHPUT' if the maximum throughput input option was chosen.

### *k. PROCESSOR DATA*

For each processor in the system, the following data is provided:

(1) ID - the processor identifier

(2) TYPE - the processor type (100 refers to I/O processors)

(3) UTILIZATION - the overall processor utilization rate

(4) EXECUTION - the utilization rate of the execution unit

(5) EXE/UTIL - the amount of execution as part of the overall utilization

(6) UTIL-EXE - the amount of communication not overlapped with execution

### *l. AVERAGE PROCESSOR UTILIZATION*

The average amount over all arithmetic processors (excluding I/O) of processor utilization during the period measurements are taken.

### *m. AVERAGE PROCESSOR EXECUTION*

The average amount over all arithmetic processors (excluding I/O) of execution unit utilization during the period measurements are taken.

### *n. AVERAGE EXECUTION / UTILIZATION RATE*

The average amount over all arithmetic processors (excluding I/O) of execution unit utilization as a portion of total utilization during the period measurements are taken.

### *o. AVERAGE NON-OVERLAPPED COMMUNICATION RATE*

The average amount over all arithmetic processors (excluding I/O) of communication not overlapped with execution during the period measurements are taken.

### *p. NORMALIZED DATA RATE*

The rate of input into the system compared to the optimum execution completion time. A value of '0' means the optimum throughput option was chosen.

### *q. SIMULATION TIME (cycles)*

The total time in cycles for the simulation to run to completion.

### *r. AVERAGE RESPONSE TIME (cycles)*

The average length of time over all measured graph instances for a graph instance to complete.

### *s. AVERAGE THROUGHPUT (Instances per Megacycle)*

The average number of data flow graphs to be completed per million cycles during the time measurements are taken..

### *t. INSTANCE LENGTH STANDARD DEVIATION*

The standard deviation of the completion time of the measured instances.

### *u. COEFFICIENT OF VARIATION*

The instance length standard deviation divided by the average response time.

### *v. I/O COMMUNICATION TIME FOR ONE GRAPH INSTANCE*

The required communication time (in cycles) for one data flow graph instance which occurs on the I/O processors..

#### *w. I/O CALCULATION TIME FOR ONE GRAPH INSTANCE*

The required calculation time (in cycles) for one data flow graph instance which occurs on the I/O processors..

#### *x. NODE COMMUNICATION TIME FOR ONE GRAPH INSTANCE*

The amount of communication time (in cycles) which is related to the graph nodes (setup latency, breakdown latency, and the time to load the instruction), excluding that which occurs on the I/O processors.

#### *y. QUEUE COMMUNICATION TIME FOR ONE GRAPH INSTANCE*

The amount of communication time (in cycles) which is related to the graph queues (reading and writing data), excluding that which occurs on the I/O processors.

#### *z. COMMUNICATION TIME FOR ONE GRAPH INSTANCE*

The required communication time (in cycles) of one data flow graph instance. This does not include the communication time and control time for input and output nodes and queues.

#### *aa. CALCULATION TIME FOR ONE GRAPH INSTANCE*

The required calculation time (in cycles) of one data flow graph instance. This does not include the calculation time for input and output nodes.

#### *ab. IDEAL CYLINDER COMMUNICATION OF ONE INSTANCE*

The amount of communication time (in cycles) which would be equally divided among the arithmetic processors. This does not include the communication which occurs on the I/O processors.

#### *ac. IDEAL CYLINDER CALCULATION OF ONE INSTANCE*

The amount of calculation time (in cycles) which would be equally divided among the arithmetic processors. This does not include the calculation which occurs on the I/O processors.

### *ad. COMMUNICATION/CALCULATION RATIO*

The ratio of the communication time to the computation time for one instance. This does not include the communication and calculation which occurs on the I/O processors.

## APPENDIX C: SAMPLE INPUT DATA FILES

**NODES.IN**

```
20

101  0  0    50000  0  6
102  0  0    50000  0  3
103  0  0   150000  0  7
104  0  0   150000  0  8
105  0  0   100000  0  4
106  0  0   100000  0  5
107  0  0  1000000  0  1
108  0  0    50000  0  1
109  0  0   400000  0  7
110  0  0  1000000  0  2
111  0  0   400000  0  8
112  0  0    75000  0  2
113  0  0  1000000  0  3
114  0  0  1000000  0  4
115  0  0  1000000  0  5
116  0  0    50000  0  8
117  0  0   800000  0  6
118  0  0    50000  0  6
119  0  0   100000  0  7
120  0  0   100000  0  8
```

**QUEUES.IN**

```
25

1    0   101  16384  16384
2    0   102  16384  16384
3  101   103  16384  16384
4  102   104  16384  16384
5  103   105  16384  16384
6  104   106  16384  16384
7  105   107   4096   4096
8  106   108   4096   4096
9  107   109   4096   4096
10  108  110   4096   4096
11  109  112   4096   4096
12  109  113   4096   4096
13  110  111   4096   4096
14  111  112   4096   4096
15  111  114   4096   4096
16  112  115   4096   4096
17  113  116      4      4
18  114  116      4      4
19  115  117   2052   2052
20  116  117      4      4
21  117  118    513    513
22  117  120    513    513
23  118  119    513    513
24  119    0    513    513
25  120    0    513    513
```

**nodes.dat**

```
22

101  0  0    50000  0  0    6   0  6  0
102  0  0    50000  0  0    3   0  3  0
103  0  0   150000  0  0    7   0  7  0
104  0  0   150000  0  0    8   0  8  0
105  0  0   100000  0  0    4   0  4  0
106  0  0   100000  0  0    5   0  5  0
107  0  0  1000000  0  0    1   0  1  0
108  0  0    50000  0  0    1   0  1  0
109  0  0   400000  0  0    7   0  7  0
110  0  0  1000000  0  0    2   0  2  0
111  0  0   400000  0  0    8   0  8  0
112  0  0    75000  0  0    2   0  2  0
113  0  0  1000000  0  0    3   0  3  0
114  0  0  1000000  0  0    4   0  4  0
115  0  0  1000000  0  0    5   0  5  0
116  0  0    50000  0  0    8   0  8  0
117  0  0   800000  0  0    6   0  6  0
118  0  0    50000  0  0    6   0  6  0
119  0  0   100000  0  0    7   0  7  0
120  0  0   100000  0  0    8   0  8  0
1    1  0        0  0  0  100 100  6  0
2    2  0        0  0  0  100 100  6  0
```

**queues.dat**

```
25

1   1    1  101  16384  16384  16384  16384  16384  0  131072  11
2   1    1  102  16384  16384  16384  16384  16384  0  131072   4
3   0  101  103  16484  16384  16384  16384  16384  0  131072   7
4   0  102  104  16384  16384  16384  16384  16384  0  131072   5
5   0  103  105  16384  16384  16384  16384  16384  0  131072   8
6   0  104  106  16384  16384  16384  16384  16384  0  131072   6
7   0  105  107   4096   4096   4096   4096   4096  0   32768   1
8   0  106  108   4096   4096   4096   4096   4096  0   32768   3
9   0  107  109   4096   4096   4096   4096   4096  0   32768  10
10  0  108  110   4096   4096   4096   4096   4096  0   32768   2
11  0  109  112   4096   4096   4096   4096   4096  0   32768   2
12  0  109  113   4096   4096   4096   4096   4096  0   32768   3
13  0  110  111   4096   4096   4096   4096   4096  0   32768   9
14  0  111  112   4096   4096   4096   4096   4096  0   32768   1
15  0  111  114   4096   4096   4096   4096   4096  0   32768   4
16  0  112  115   4096   4096   4096   4096   4096  0   32768   8
17  0  113  116      4      4      4      4      4  0    3200   2
18  0  114  116      4      4      4      4      4  0    3200   8
19  0  115  117   2052   2052   2052   2052   2052  0   16416   6
20  0  116  117      4      4      4      4      4  0    3200   7
21  0  117  118    513    513    513    513    513  0    4104   7
22  0  117  120    513    513    513    513    513  0    4104   9
23  0  118  119    513    513    513    513    513  0    4104   7
24  2  119    2    513    513    513    513    513  0    4104   2
25  2  120    2    513    513    513    513    513  0    4104   3
```

# APPENDIX D: SAMPLE RUN OF PROGRAMS

This appendix outlines the general procedure for running a simulation session with the programs in this thesis and [Ref. 4]. Input and output files will be indicated in **bold**. Commands required to run programs will be indicated in *italics*. Consult the user's manual (Appendix B) for detailed descriptions of the program input and output files. Note that all output files are opened for writing (except **stats.out**) during program execution. This means that file names must be modified when running successive iterations of the same program or data will be lost.

1. Modify the **NODES.IN**, **QUEUES.IN**, **PROCS.IN**, and **CHAINS.IN** files for the graph to be analyzed.

2. Run the node allocation program by typing *schedule* and entering the proper input data. This program will generate the **ORDER.IN** file and the **cylinder.out** file.

3. Run the mapper program by typing *map* and entering the proper input data. This will generate the **cylinder.dat** file and other descriptive output files.

There are now several different steps to be taken, depending on whether it is desired to analyze the FCFS or RC technique and dependent on whether wrap-around or no wrap around is desired. Each of the techniques and variations will be discussed separately.

## A. FCFS

Modify the **nodes.dat**, **queues.dat**, and **machine.cfg** files for the graph to be analyzed. The **cylinder.dat** file must also be present for the program to run, although it is not necessary to be concerned about the wrap-around or no wrap-around option since the simulator does not use this file for FCFS. Run the simulator program by typing *simulate* and enter the proper input data.

## B. RC TECHNIQUES

For the various RC techniques, synchronization arcs must be generated before the simulator program is run.

### 1. Start-after-start (SAS) or start-after-finish (SAF) without binding nodes to processors.

The **machine.cfg** file must contain a zero for each processor assigned.

#### *a. No wrap-around*

Use the **cylinder.dat** file from the mapper program.

#### *b. Wrap-around:*

The cylinder.out file must be renamed to cylinder.dat. Ensure the cylinder.dat file is renamed first, or it will be overwritten.

Run the synchronization arc program by typing *generate*. Select the technique desired. If it is desired to generate another set of synchronization arcs for a different technique (i.e., SAF arcs are generated, and SAS is now desired) the **queues.dat** file must be renamed (to something appropriate, e.g., **queues.SAF**) and the **oqueues.dat** file renamed to **queues.dat**. The generate program can now be run for the new technique.

### 2. Start-after-start (SAS) or start-after-finish (SAF) with binding nodes to processors.

For this technique, the **machine.cfg** file must now contain a number for each processor assigned (i.e., 1, 2, etc.). The same rules apply as above for generation of arcs for wrap-around and no wrap-around techniques.

## C. SIMULATIONS

For the simulations it is important to maintain the proper input files. Ensure the **queues.dat** file matches the appropriate **cylinder.dat** file, i.e., wrap or no wrap and that the **machine.cfg** file corresponds to the desired binding or no binding condition. As an example, say the input files were originally named (after generating the synchronization arcs) **queues.SAFnW** (no wrap), **queues.SAFbnW** (bound, no wrap),

**queues.SAFnW** (with wrap), **queues.SAFbW** (bound, with wrap), **cylinder.W** (with wrap), **cylinder.nW** (no wrap), **machine.cfgb** (bound), and **machine.cfgnb** (not bound). In order to run a simulation for the no wrap, non-bound configuration, the files, **queues.SAFnW**, **cylinder.nW**, and **machine.cfgnb** must be renamed to **queues.dat**, **cylinder.dat**, and **machine.cfg**. The simulator program can now be run by typing *simulate*. Remember that the three files named above must be renamed again in order to simulate new RC techniques.

## LIST OF REFERENCES

1. Shukla, S.B., Little, B.S., and Zaky, A., "A Compile-Time Technique for Controlling Real-Time Execution of Task-Level Data Flow Graphs," presented at the 1992 International Conference on Parallel Processing, St. Charles, Illinois.

2. Cross, D. M., *Usefullness of Compile-Time Restructuring of LGDF Programs in Throughput-Critical Applications*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.

3. Bell, H.A., *A Compile-Time Approach For Chaining and Execution Control in the AN/UYS-2 Parallel Signal Processing Architecture*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1992.

4. Cross, D.M., Shukla, S.B., and Zaky, A., *Revolving Cylinder Analysis: A Technique for Restructuring of Large Grain Data Flow Graphs Representing Throughput-Critical Applications*, Naval Postgraduate School Technical Report NPS-EC-93-015, September 1993.

5. Akin, C., *Efficient Scheduling of Real Time Compute-Intensive Periodic Graphs on Large Grain Data Flow Multiprocessor*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1993.

## INITIAL DISTRIBUTION LIST

| | | |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, VA 22314-6145 | 2 |
| 2. | Dudley Knox Library, Code 52<br>Naval Postgraduate School<br>Monterey, CA 93943-5101 | 2 |
| 3. | Chairman, Code EC<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5121 | 1 |
| 4. | Prof. Shridhar B. Shukla, Code EC/Sh<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5121 | 2 |
| 5. | Prof Amr Zaky, Code CS/Za<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, CA 93943-5118 | 2 |
| 6. | Mr. David Kaplan<br>Naval Research Laboratory<br>4555 Overlook Avenue, SW<br>Washington, D.C. 20375-5000 | 1 |
| 7. | Mr. Richard Stevens<br>Naval Research Laboratory<br>4555 Overlook Avenue, SW<br>Washington, D.C. 20375-5000 | 1 |
| 8. | Mr. Paul J. Hays<br>Mail Stop 473<br>Langley Research Center<br>Information Systems Division<br>Hamton, VA 23681-0001 | 1 |
| 9. | LT John P. Cardany, USN<br>113 Mervine St.<br>Monterey, CA 93940-6205 | 1 |
| 10. | Mr. W. John Pohl<br>1710 Springhill Ct.<br>Louisville, KY 40223 | 1 |